THESIS

SYMBOLIC EXECUTION OVER NATIVE X86

by

Michael Hom

June 2012

Thesis Advisor:	Chris S. Eagle
Second Reader:	George W. Dinolt

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 4–5–2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* 2010-03-26—2012-03-26

4. TITLE AND SUBTITLE Symbolic Execution Over Native x86	**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Michael Hom	**5d. PROJECT NUMBER** **5e. TASK NUMBER** **5f. WORK UNIT NUMBER**
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Navy	**10. SPONSOR/MONITOR'S ACRONYM(S)** **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.IRB Protocol Number: N/A

14. ABSTRACT

Current approaches to program analysis largely rely on the use of an intermediate language to derive intermediate representations of source code or binaries under evaluation. This can simplify semantics when dealing with a complex instruction set such as the Intel Industry Standard Architecture (ISA) instruction set. However, a question that remains is whether these intermediate languages truly retain semantic fidelity or whether elements of the ISA instruction set get lost in translation. This thesis describes a framework that is being developed at NPS that accomplishes symbolic execution without the use of an intermediate language and symbolically executes ELF and WinPE binary programs over the native x86 ISA instruction set, and specifically discusses an approach to describing state mathematically using a formal algebra.

15. SUBJECT TERMS

Program Analysis, Symbolic Execution, Theorem Proving, x86, SMT, SAT

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	107	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SYMBOLIC EXECUTION OVER NATIVE X86

Michael Hom
Civilian, Department of the Navy
B.S., University of California, San Diego, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2012

Author:	Michael Hom
Approved by:	Chris S. Eagle Thesis Advisor
	George W. Dinolt Second Reader
	Peter J. Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Current approaches to program analysis largely rely on the use of an intermediate language to derive intermediate representations of source code or binaries under evaluation. This can simplify semantics when dealing with a complex instruction set such as the Intel Industry Standard Architecture (ISA) instruction set. However, a question that remains is whether these intermediate languages truly retain semantic fidelity or whether elements of the ISA instruction set get lost in translation. This thesis describes a framework that is being developed at NPS that accomplishes symbolic execution without the use of an intermediate language and symbolically executes ELF and WinPE binary programs over the native x86 ISA instruction set, and specifically discusses an approach to describing state mathematically using a formal algebra.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

BDD	Binary Decision Diagram
CFG	Control Flow Graph
COTS	Commercial-Off-the-Shelf
DART	Directed Automated Random Testing
GOTS	Government-Off-The-Shelf
IDA	Interactive Disassmbler
IL	Intermediate Language
IR	Intermediate Representation
ISA	Instruction Set Architecture
LFCS	Labeled Formal Concurrent System
LLVM	Low Level Virtual Machine
NPS	Naval Postgraduate School
SAGE	Scalable Automated Guided Execution
SAT	Satisfiability
SCADA	Supervisory Control and Data Acquisition
SMT	Satisfiability Modulo Theory
USG	United States Government

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

I would like to acknowledge the following people:

Chris Eagle for being a great thesis advsior and great mentor to me and countless others. Most of us are where we are because of your efforts to teach us and encourage deep thinking on technical subjects.

George Dinolt for his mentorship and guidance in my thesis topic and general academia. Our discussions are interesting and your insight into the history of our field provides context that few understand.

Simson Garfinkel for his mentorship during my time at NPS. Your dedication to your students is uncanny. The time I spent learning from you during my internship provided incredible insight to the world of forensic research and how to think differently to solve hard problems.

Cynthia Irvine for managing the Scholarship For Service program at NPS. You are providing an experience and subsequent opportunities for people that are hard to come by.

Valerie Linhoff for getting me through all phases of NPS and for being incredibly patient with me.

My wife for being an incredible person and making sure I stayed on track.

My family for always being the anchor I need, whether I know it or not.

This material is based on work supported by the National Science Foundation under Grant DUE - SFS Scholarships.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

The ability to monitor the behavior of a computer program during execution has become a critical capability in computer security. The potential of being able to precisely determine the behavior of a computer program augments the efforts of a broad range of computer security research areas including:

- Memory Analysis
- Software Test Case Generation
- Software Vulnerability Assessments
- Software Optimization
- Software Verification

Symbolic execution is an abstraction of the concept of testing every possible input fed to a program by tracking symbolic values instead of actual values [1, 2]. Coupling symbolic execution and static program analysis allows us to derive a wealth of information about a binary program and offer insight into the behavior of the binary program and its supposed functionality. With symbolic execution, one can reason about all possible inputs that a binary program might accept when executing. It is not realistically feasible to test every possible input that can be generated and fed into a program. Conversely, it would be very useful to try to test every input possible instead of generating a subset of inputs to fuzz test [3–6] a program.

To generalize, we utilize symbolic execution to attempt to test specific properties of a program we are interested in. As such, employing symbolic execution to perform program analysis requires a precise representation of the constraints that will ultimately provide inputs that lead to satisfied test cases. For example, if the goal of performing symbolic execution for program analysis is to determine inputs that will result in stack-based or heap-based buffer overflows, then it is necessary to know the exact constraints on the inputs that led to the overflow.

Evaluating the flow of programs to determine whether a calculated input leads to a program that can be framed as a decision problem. That calculated input consists of a set of constraints that can lead to a potential vulnerability in that binary. To solve these decision problems, these constraints must be formally represented and supplied to an evaluation engine that can accurately determine whether a given path reaches a potentially vulnerable function.

Capturing appropriate constraints is a critical issue when solving decision problems and associated satisfiability because these constraints capture the state of a program at that exact point in execution. The language that describes the constraints for any program must be detailed enough to encompass what an input constraint can be for *any* given program. Simultaneously, the language must be concise enough so that the efficiency of solving satisfiability is maximized.

1.1 Research Questions

This thesis poses a primary question that is addressed in Chapter 4: Can an algebra be developed to mathematically describe the state of binary code during execution? More specifically, can this algebra describe the behavior of each instruction in the x86 instruction set, pre- and postconditions of a basic block within binary code, and the overall state of the binary code under execution?

To answer this question, an algebraic framework will be developed to describe how the state of a program under execution can be represented as a set of constraints that will subsequently be tested for satisfiability in reaching a point of execution.

1.2 Significant Findings

Using native x86 Intel architecture, a framework was constructed to accurately model the state of binary code under execution through an algebra. Through this algebra, a basic methodology was constructed to mathematically represent program state at the instruction, basic block, and program level.

1.3 Thesis Structure

This thesis is organized as follows:

- Chapter 2 gives a background of symbolic execution, discusses some applications of symbolic execution in computer security research, and gives a survey of related work.
- Chapter 3 describes the development of the program analysis framework that utilizes constraint representation.
- Chapter 4 discusses the algebra that is used to mathematically describe the state of binary code under execution, and as a result, path constraint decision problems.
- Chapter 5 provides conclusions drawn from the previous chapters and recommended future work.

CHAPTER 2: Background and Related Work

In this section, we provide the motivation behind the project, discuss some key concepts to provide a context for our approach, and provide a survey of related work in the field.

2.1 Motivation

Computing is now a ubiquitous part of today's society and is deeply intertwined in day-to-day operations of the world. The majority of devices used on an everyday basis are powered by computers of various types and the underlying software that allows people to interact with them—whether they be desktop computers, smart phones, television, cars, or even household appliances. The importance of computing software has reached such a critical level that real-time systems such as Supervisory Control and Data Acquisition (SCADA) systems are the driving force behind large-scale industrial systems—such as power centers, water treatment plants, and gas pipelines—that operate continuously. Along with ever increasing usage and requirements of software, there has been a substantial growth of complexity [7].

With the many solutions software provides, there are just as many new problems that arise. Software bugs are prevalent through the lifecyle of any given piece of software and finding them can prove to be an arduous process. The necessity for finding bugs in the context of information security is of critical importance because of the dire consequence of not fixing them, whether it be economic, social, or worse.

There has been a growing trend for malicious software to be spread through the internet, whether it is to spread indiscriminately, or to be aimed at specific targets [8–10]. Researchers who encounter these pieces of malicious software, or malware, in the wild have the job of analyzing them to understand how they behave and to possibly create countermeasures. There are two primary approaches to performing this analysis: static analysis and dynamic analysis [11].

Static analysis is the act of analyzing a piece of software, whether it is source code or a compiled binary without actually executing the program. With compiled binary programs, the analysis typically involves disassembling the program to recover the program's assembly code instructions, or to go further and attempt a decompilation of the program by turning the disassembly back into a high-level language (e.g., C/C++). Unfortunately, there are problems with trying to fully decompile a program and Eagle [12] describes them well:

- The compilation process is lossy.
- Compilation can be a many-to-many operation.
- Decompilers are very language, compiler, and library dependent.
- A nearly perfect disassembly capability is needed in order to accurately decompile a binary.

Working with the disassembly may be more attractive than trying to decompile a program when performing an analysis since the disassembly is easier to obtain with some degree of accuracy but the problem remains that disassembly, by nature, is imperfect.

With tools like IDA Pro [13], a program analyst can disassemble a compiled binary and proceed to perform a manual static analysis of that program to ascertain the program's behavior and possibly determine vulnerabilities that exist within the program. Depending on the skill level and experience of the analyst, this can be a very effective process for a single program, but may take a long time.

Dynamic analysis is the act of allowing a program to execute and within a carefully controlled environment, often known as a sandbox, while the analyst observes the behavior of the program through system instrumentation. In this scenario, the analyst instruments a binary in a sandbox environment such as a virtual machine to observe program behavior at several levels: the process level, the registry level, the network level, or the storage level, depending on theplatform. This offers some advantages over static analysis, most notably, savings in time to understand some of the behaviors of the program. Instead of poring over lines of disassembly, an analyst can execute a program and observe changes to the program environment as the program runs.

Dynamic analysis can certainly provide cost-savings with time but may be subject to other disadvantages. Some malicious software will check for the presence of instrumentation or query the environment in which it is operating. If the software is being run under debugger control, for instance, the process may choose to exit on its own before any salient behavior can be observed. Such techniques are called anti-debugging techniques [14, 15].

2.2 Disassembly and Intermediate Language Limitations

2.2.1 Disassembly Limitations

By nature, disassembly of a program binary is imperfect. These imperfections can be categorized as follows [16]:

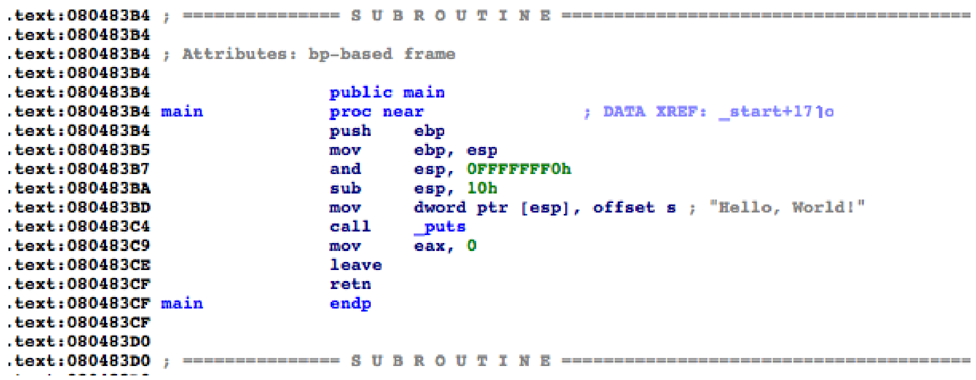
- False Positives: Misidentified instructions
- False Negatives: Non-disassembled instructions

A disassembler's job is to determine whether an instruction should be disassembled or not based on whether it is referenced by another instruction [12]. The behavior of these instructions will determine how the program's control flow is represented during the disassembly of a program. The accuracy and precision of a disassembler is predicated on its ability to recover data type information, distinguishing instructions from data, platform dependencies of the disassembler, and library function identification. The ability to accurately and precisely address these critical areas of a program are what drive results of a good disassembler [17]. Impediments to accurate disassembly include the following [16]:

- Data embedded in the code regions
- Variable instruction size
- Indirect branch instructions
- Functions without explicit `CALL` sites within the executable's code segment
- Position Independent Code (PIC) sequences
- Hand crafted assembly code

Further, techniques exist that attempt to counter a disassembler's ability to properly disassemble a program. These *anti-disassembly* techniques are methods that cause a disassembler to generate an incorrect or incomplete disassembly listing. Generally, programs that make use of anti-disassembly techniques contain instructions that, when interpreted by the disassembler, effectively confuse it and cause it to misidentify the correct instruction bytes at the time of disassembly [18]. This can lead to improperly identified instruction sequences, incomplete instructions sequences, or cause the disassembler to miss sections of code altogether.

Malware authors commonly try to incorporate anti-disassembly techniques to thwart analysis of malicious software through obfuscation techniques. These techniques include, but are not limited to self-modifying code and virtual machine based obfuscation [14,19]. These techniques can obscure real code with code that visually may or may not make sense to an analyst but ultimately offers code that can lead to an incorrect analysis. A comparison can be made with Figure 2.1 and Figure 2.2. Figure 2.1 shows what IDA Pro displays of an unobfuscated binary (hello world) after initial analysis, whereas Figure 2.2 shows what IDA Pro displays of an obfuscated binary (hello world) after initial analysis. In this case, the obfuscation technique comes from using the Ultimate Packer for eXecutables (UPX) [20].



```
.text:080483B4 ; =============== S U B R O U T I N E =======================================
.text:080483B4
.text:080483B4 ; Attributes: bp-based frame
.text:080483B4
.text:080483B4                 public main
.text:080483B4 main            proc near               ; DATA XREF: _start+17↑o
.text:080483B4                 push    ebp
.text:080483B5                 mov     ebp, esp
.text:080483B7                 and     esp, 0FFFFFFF0h
.text:080483BA                 sub     esp, 10h
.text:080483BD                 mov     dword ptr [esp], offset s ; "Hello, World!"
.text:080483C4                 call    _puts
.text:080483C9                 mov     eax, 0
.text:080483CE                 leave
.text:080483CF                 retn
.text:080483CF main            endp
.text:080483CF
.text:080483D0
.text:080483D0 ; =============== S U B R O U T I N E =======================================
```

Figure 2.1: Disassembly of the Main Function to a Hello World Program (Taken from IDA Pro)

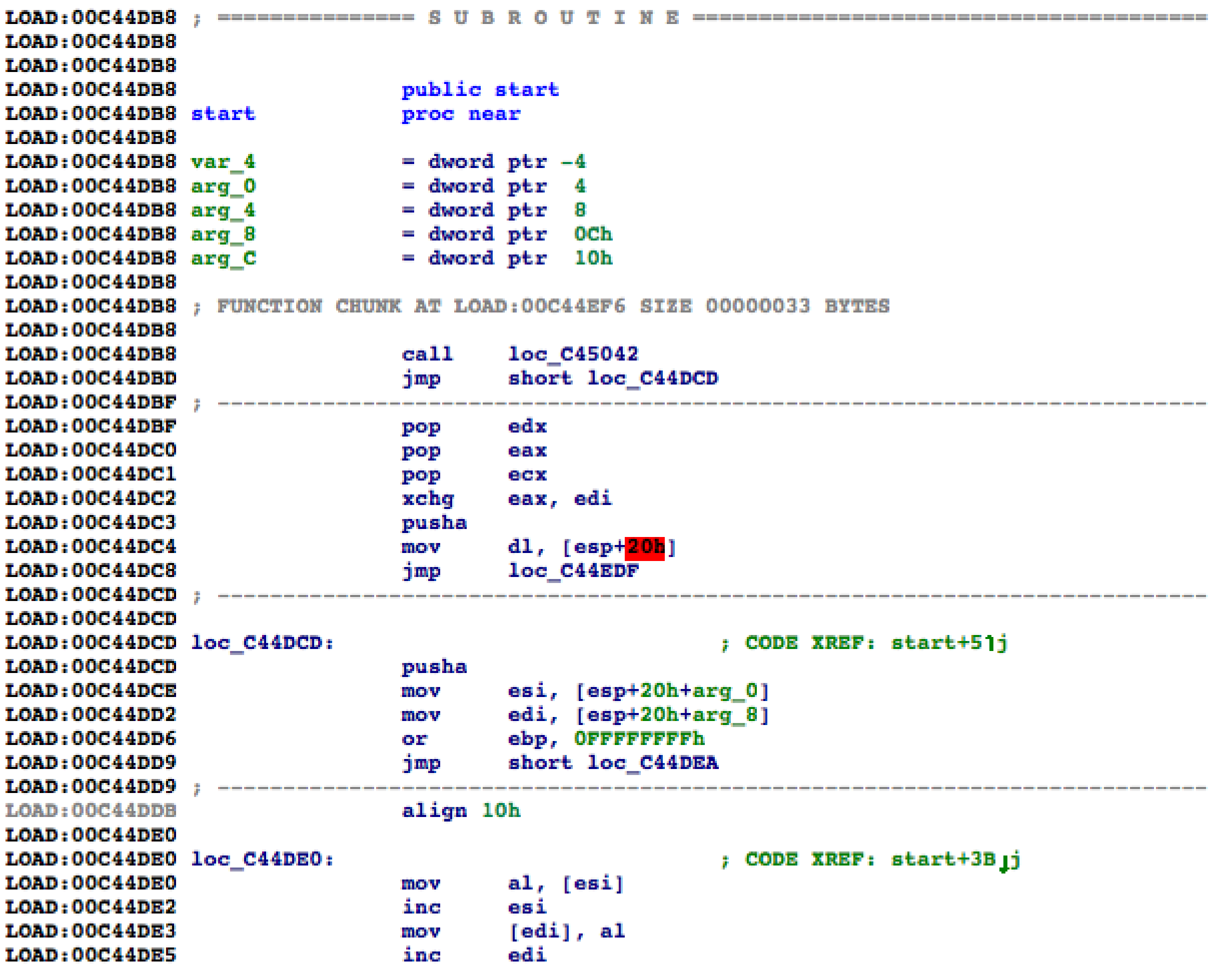
Additional comparisons can be made when inspecting a control flow call graph of both "hello world" iterations.

2.2.2 Intermediate Language Limitations

Intermediate languages are a ubiquitous part of program analysis because of their use in compilers [21]. One the main attractions of intermediate languages is how they:

- Reduce semantic complexity during analysis
- Allow code transformations for different architectures from one source
- Allow code analysis optimization

Intermediate language (IL) frameworks like LLVM (formerly standing for Low Level Virtual Machine) [22] are popular because of their ability to retain and honor the semantics of a piece of source code [21], though the extent to which each IL framework honors the original semantics differs based on the depth and complexity of the framework itself. This is advantageous for compilers at various stages of compilation since it allows developers to write compilers that can be used across different OS platforms. For program analysis frameworks [23, 24], an IL can simplify the semantic complexity of a CPU instruction set by abstracting away from mnemonic



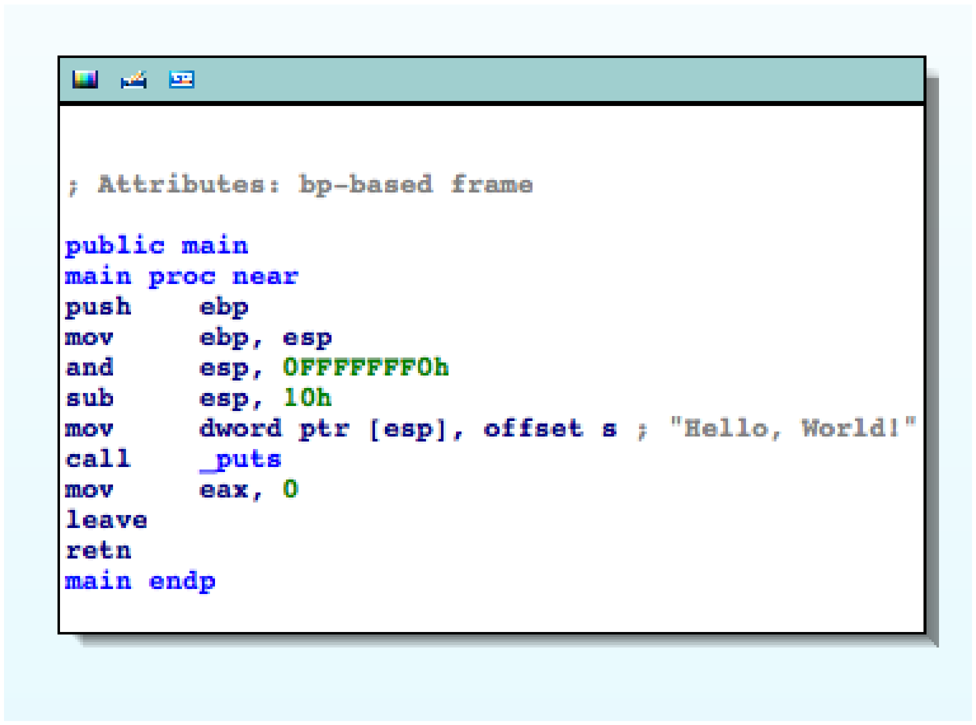
```
LOAD:00C44DB8 ; =============== S U B R O U T I N E =======================================
LOAD:00C44DB8
LOAD:00C44DB8
LOAD:00C44DB8                 public start
LOAD:00C44DB8 start           proc near
LOAD:00C44DB8
LOAD:00C44DB8 var_4           = dword ptr -4
LOAD:00C44DB8 arg_0           = dword ptr  4
LOAD:00C44DB8 arg_4           = dword ptr  8
LOAD:00C44DB8 arg_8           = dword ptr  0Ch
LOAD:00C44DB8 arg_C           = dword ptr  10h
LOAD:00C44DB8
LOAD:00C44DB8 ; FUNCTION CHUNK AT LOAD:00C44EF6 SIZE 00000033 BYTES
LOAD:00C44DB8
LOAD:00C44DB8                 call    loc_C45042
LOAD:00C44DBD                 jmp     short loc_C44DCD
LOAD:00C44DBF ; ---------------------------------------------------------------------------
LOAD:00C44DBF                 pop     edx
LOAD:00C44DC0                 pop     eax
LOAD:00C44DC1                 pop     ecx
LOAD:00C44DC2                 xchg    eax, edi
LOAD:00C44DC3                 pusha
LOAD:00C44DC4                 mov     dl, [esp+20h]
LOAD:00C44DC8                 jmp     loc_C44EDF
LOAD:00C44DCD ; ---------------------------------------------------------------------------
LOAD:00C44DCD
LOAD:00C44DCD loc_C44DCD:                             ; CODE XREF: start+5↑j
LOAD:00C44DCD                 pusha
LOAD:00C44DCE                 mov     esi, [esp+20h+arg_0]
LOAD:00C44DD2                 mov     edi, [esp+20h+arg_8]
LOAD:00C44DD6                 or      ebp, 0FFFFFFFFh
LOAD:00C44DD9                 jmp     short loc_C44DEA
LOAD:00C44DD9 ; ---------------------------------------------------------------------------
LOAD:00C44DDB                 align 10h
LOAD:00C44DE0
LOAD:00C44DE0 loc_C44DE0:                             ; CODE XREF: start+3B↓j
LOAD:00C44DE0                 mov     al, [esi]
LOAD:00C44DE2                 inc     esi
LOAD:00C44DE3                 mov     [edi], al
LOAD:00C44DE5                 inc     edi
```

Figure 2.2: Part of a Disassembly to a Hello World Program Packed with the Ultimate Packer for eXecutables (UPX) (Taken from IDA Pro)

operation details. However, the design of an IL language leads to issues including the following [21]:

- Depth level—In particular, how machine-dependent it is
- Structure of the language
- Expressiveness—How accurately does it honor original source program semantics
- The types of transformations that can be performed on the IL representation of a source program

It may be the case that IL frameworks dilute the semantics from the original source. An IL and its corresponding translators must be sufficiently rich to express all the behaviors in an instruction's operation. This can be rather difficult since the behavior of x86 can occasionally



```
; Attributes: bp-based frame

public main
main proc near
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
mov     dword ptr [esp], offset s ; "Hello, World!"
call    _puts
mov     eax, 0
leave
retn
main endp
```

Figure 2.3: A Simple Hello World Graph (Taken from IDA Pro)

be undocumented or documented in an ambiguous manner, and not necessarily mathematically rigorous. Given this, our approach to analyzing a program is to do away with an IL altogether and consider the native x86 assembly as it is encountered during symbolic execution. This may lead to a higher level of complexity when reasoning over native Intel x86 architecture with no abstraction.

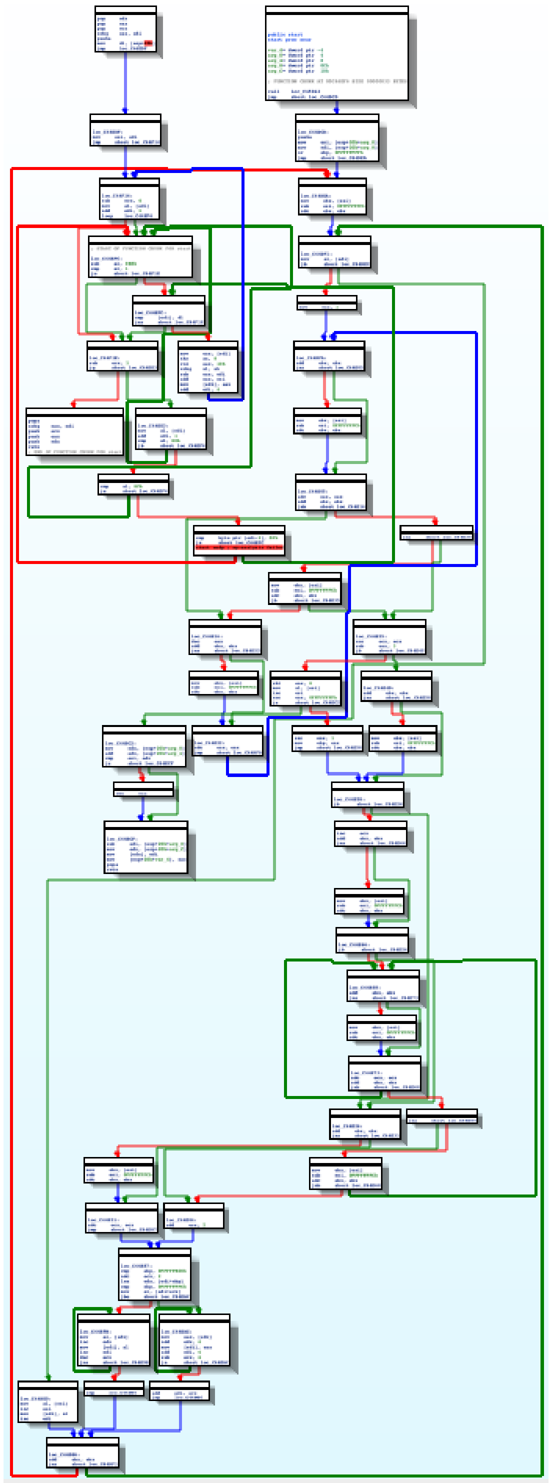


Figure 2.4: A Decidedly More Complicated Graph of Hello World with UPX (Taken from IDA Pro)

2.3 Symbolic Execution

Dynamic analysis is an attractive approach to performing program analysis because it offers the ability to monitor code and program state as it executes. Symbolic execution is an abstraction of normal program execution, where a program is symbolically executed not on a set of sample concrete inputs but rather a set of classes of inputs [2]. This can be interpreted as executing over a set of variables that eventually become bounded to sets of values, which leads to demonstrating desired program properties. Contemporary symbolic execution is a form of dynamic analysis used to constrain the reasoning of a symbolically executed program to logical constraints that are then fed into a model to solve for satisfiability [1].

Symbolic execution is a powerful tool to use when performing dynamic program analysis but there are advantages and disadvantages to the approach. Symbolic execution, by its nature has no runtime context. This is because the initial input, when symbolically executing a program, is unconstrained. In other words, the data domain is unbounded. This allows a complete and exhaustive exploration of all runtime state space in a given program. Since the input is unconstrained, it can be “anything”, and code dependent on the symbolic input can be used to build logical path constraints that can be used to explore runtime states that may be of interest. In other words, symbolic execution operates on all possible sets of values, whereas normal (concrete) execution operates only on a single possible set of concrete values that have been defined by the program’s author or inputted by the program user [25].

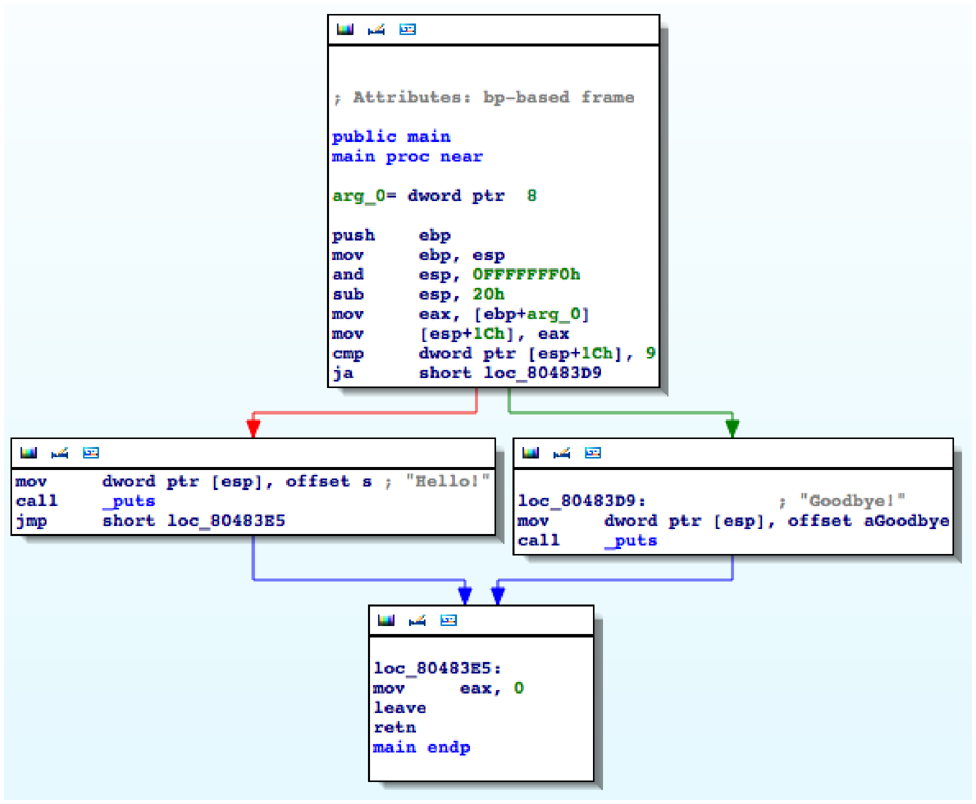
The first step in symbolic execution is to generate a Control Flow Graph (CFG). A CFG is an abstract representation of all possible paths that can be traversed through a program during its execution using a directed graph, or digraph. In a CFG, each node in the digraph represents a basic block of program instructions. A basic block is defined by Aho, Sethi, and Ullman [26]:

- The block of instructions has one entry point.
- The block of instructions has one exit point.
- Each instruction in a basic block executes before all succeeding instructions in later positions in that same block.
- No other instruction executes between two instructions in the sequence.
- The first instruction in a basic block is known as a *leader*. The rule for finding leaders are:
    1. The first instruction is a leader.
    2. Any instruction that is the target of a conditional or unconditional jump is a leader.
    3. Any instruction that immediately follows a conditional or unconditional jump is a

leader.

- A basic block consists of a leader and all instruction up to, but not including, the next leader or the end of the program.

In the CFG, directed edges represent jumps in the program execution, each edge is a boolean “truth value” for the condition. Figures 2.5, 2.3, and 2.4 illustrate what a CFG can look like.



```
; Attributes: bp-based frame

public main
main proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
mov     eax, [ebp+arg_0]
mov     [esp+1Ch], eax
cmp     dword ptr [esp+1Ch], 9
ja      short loc_80483D9
```

```
mov     dword ptr [esp], offset s ; "Hello!"
call    _puts
jmp     short loc_80483E5
```

```
loc_80483D9:                ; "Goodbye!"
mov     dword ptr [esp], offset aGoodbye
call    _puts
```

```
loc_80483E5:
mov     eax, 0
leave
retn
main endp
```

Figure 2.5: A Simple Example of a CFG (Taken from IDA Pro)

In the context of finding bugs in a program, symbolic execution helps determine inputs that lead to memory corruption or unexpected paths. The reasoning here is we want to find inputs that will eventually lead to a certain point in the CFG so that we can backtrace to see what *concrete* inputs will cause us to go down that particular path, whether a memory corruption occurs or not. The CFG is used in the analysis to determine the path or paths that will indeed result in memory corruption of a given program. More precisely, we want to derive the set of classes of inputs that will lead us to reachable paths in a programs execution that result in memory corruption, by reasoning over the symbolic values assigned to the program being tested. When reasoning over the symbolic values during analysis, there are conditions (e.g., concrete values or sets of concrete values) that are derived as a result of a path being taken. These path conditions, or path constraints, are the Boolean statements that form the conditional statements encountered at each node. Essentially, each set of path constraints can be associated with each digraph edge on a given path.

When a suitable path has been determined, the corresponding set of path constraints are used to determine path reachability with a theorem prover. That is, the set of path constraints are the set of classes of inputs that may lead to a memory corruption if they can be proven satisfiable with a theorem prover.

The approach to using symbolic execution to generate logical path constraints and solve the boolean satisfiability of the constraint formula is complete and exhaustive, but it is an NP-complete problem making it infeasible to be able to completely solve all possible cases of satisfiability [27]. This is because any non-trivial path constraint may consist of so many terms that iterating through every possible assignment for the Boolean variables would never complete. This is part of the state, or path, explosion problem that exists for symbolic execution. Any non-trivial program has an intractable number of execution paths. Because of this, there are many approaches to dealing with path selection as well as slicing portions [28] of the program that will be symbolically executed for analysis. Despite the difficulties of iterating through all possible results, solving such path constraints has been made tractable, in some cases, with the use of theorem provers, such as satisfiability (often abbreviated to SAT) solvers, in a reasonable amount of time[1] [27, 29].

Existing symbolic execution engines try to take into account external system and library calls made by the binary but their respective implementations are limited by design complexity or scope complexity [30]. Our engine analyzes the *system of software* that is associated with a binary program. This includes:

- The binary under analysis
- Any dynamic library code used by the program

Below, we describe related work on other program analysis platforms and work done in reducing the problem of state explosion.

BITBLAZE. BitBlaze [23] is a platform of binary analysis tools that combine different approaches of analysis. The platform is made up of three primary components: *Vine*, *TEMU*, and

[1]There are, of course, path constraints that still require more time to solve than there are years in the Universe.

Rudder. Vine is the static analysis component of the platform that raises the binary machine code to an IL. TEMU is a program emulator that performs extensive traces of a program and outputs the trace to a file for later analysis. Rudder is used to provide both concrete and symbolic execution capabilities. In order to translate binary code into the IL, a disassembly output is created then passed to *VEX*, a third party library that is part of the Valgrind dynamic instrumentation tool [31]. Finally, the VEX IL output is translated to Vine IL. TEMU is a whole-system emulator based on QEMU [32] and used to instrument a binary for analysis. Rudder performs as a TEMU plugin and performs much of the navigation during binary instrumentation. It determines whether instructions should be symbolically or concretely executed, performs path selection at each branch, and encodes the path constraints for SAT solving.

BitBlaze was developed to analyze x86 binaries but work has begun to expand it to other platforms with the use of their Vine IL. It has been used academically and in industry to find bugs in binaries.

KLEE. Klee [24] is a symbolic execution platform that was redesigned from another tool originally named EXE [33]. Klee was designed to perform deep checking of applications and maximize code coverage across diverse classes of programs written in C. Klee is built upon LLVM to help with the symbolic execution of the intermediate representation (IR) of programs under testing. A program considered for testing is compiled into LLVM bytecode, the IL that provides the IR for analysis. Klee then runs on the IR to perform the symbolic execution using STP, an SMT-based constraint solver, [34] for solving path constraints.

Klee has been used both academically and in industry to conduct code coverage [24]. It has also been extended by users and subsequently had some of these extensible features integrated into the Klee core platform.

CODESURFER/X86. CodeSurfer/x86 [35–37] is a platform that utilizes abstract interpretation [38], a generalized case of symbolic execution to recover an IR of an executable. The recovered IR includes CFGs, information about the program's variables, and data dependencies.

According to the Grammatech website [37], CoderSurfer/x86 is a research prototype and not a commercial product, but is known to be used for academic research.

DART. DART (Directed Automated Random Testing) [39] is a symbolic execution platform that aims to systematically test all feasible paths of a program. One if its key values is the ability to reduce imprecision and computation time in the symbolic execution on an application under test by using concrete values at branch instructions that are of interest. DART was originally implemented at Bell Labs by Patrice Godefroid.

SAGE. SAGE (Scalable, Automated, Guided Execution) [4,5] is a platform that uses symbolic execution to perform x86 instruction-level tracing and emulation. The main difference here, and hence why the authors refer to this analysis process as fuzzing, is because it uses a well-formed input as opposed to assuming an unconstrained input. This well-formed input is used to derive path constraints as each branch instruction is executed. SAGE is based on DART [39] but optimized for large applications.

SAGE is developed and maintained by Microsoft and is heavily used to find bugs in development and commercial releases of their software.

PIN and DynamoRIO. Pin [40] and DynamoRIO [41] are both tools that allow dynamic binary instrumentation of a piece of software under test. This allows analysts to execute a program and perform transparent analysis. This means that you can, for instance, inject analysis code into a program during runtime execution. The platforms operate on single user-level processes and do not differentiate the runtime execution environment of each respective tool and the program under instrumentation.

Pin and DynamoRIO were both developed under industry and academic collaboration and have been applied to both fields.

VALGRIND. Valgrind [31] is a dynamic binary instrumentation platform, similar to Pin and DynamoRIO but is specific to UNIX/Linux variant OS environments, where Pin and DynamoRIO are available for UNIX/Linux and Windows OS environments.

Valgrind has been used in both academic and industry work.

2.4 Computational Complexity Issues

With all these criteria that go into analyzing a binary, it is important to note that the computational complexity grows when dealing with the state explosion problem associated with program analysis. In order to provide some context, consider the state space of a program under execution. It is possible, theoretically, to explore all states in order to derive path constraints and check properties in question. In practice, this is realistically unfeasible because of the number of transitions that can occur at each conditional branch instruction. This challenge is known as the state explosion problem and significantly complicates program analysis [42].

To provide some more context to this problem, imagine looking at a control flow graph of Microsoft's *word.exe* executable. This binary can be considered fairly complex because of the number of calls to other libraries—we will call them libaries set *A*—it needs to function as well as other libraries—we will call them libraries set *B*—that *A* must call. Under evaluation, word.exe can execute hundreds of library calls on top of executing a number of functions in the start up procedure. These functions can lead to branch instructions that can result in a large number of realized transitions. If the program were to theoretically continue generating parallel transitions through its execution lifecycle, we can see that the number of states would exponentially increase. If were to attempt to explore each state in either a breadth-first or depth-first manner, we can see this become an intractable task.

Aside from the intractability issue associated with exhaustively exploring the state space of a program, we also need to consider current technology when trying to tackle this problem. Using the *word.exe* example above, we also need to consider resources utilized during state exploration. If *word.exe* generates T number of threads and we are trying to evaluate this program symbolically with C number of processors, we can assume that T/C threads will be mapped onto each processor, assuming each processor receives an equal number of threads. Given a normal producer/consumer relationship among the threads (i.e., the threads can share common process resources), and assuming an amount M of system memory that is allocated per thread, we will require (T*M)/C amount of memory. If we consider C to be a very small number compared to T, and T can continually grow during a program's execution (i.e., T can theoretically approach infinite, and C stays finite), then the amount of memory required explodes and also tends towards infinite. This, however, is limited by current technology where the amount of memory can tend towards infinite but is bounded by address space limits. For example, a 32-bit x86 process is bounded by a 4GB process memory address space.

Figure 2.6 shows a simple program visualized by its CFG. It is interesting to note how complex a seemingly simple program can be after it has gone through the compilation process. Appendix A and Appendix B provide the source and disassembly listing from IDA Pro, respectively for further reference.

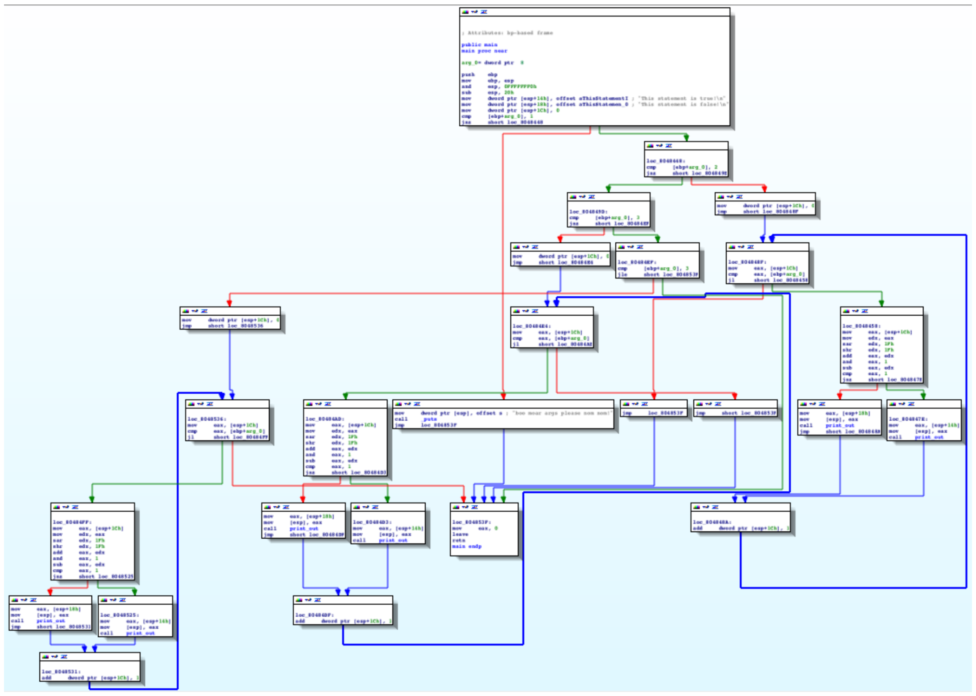


Figure 2.6: A Functionally Simple Program Visualized by Its CFG (Taken from IDA Pro)

2.4.1 State Explosion Research

Scalability issues illustrated above has been a topic of research for quite some time [43–46] and is an obstacle when dealing with the complexity of program analysis.

The task of complete code coverage in program analysis is realistically impossible as the state space of a program is generally too large to be exhaustively explored. Because of this fundamental problem, there has been research related to managing and dealing with state explosion that attempts to create precision in code coverage and program slicing [28] to try to reduce the state space when analyzing source code or binaries.

Boonstoppel [47] proposed a technique for detecting and pruning large numbers of redundant paths while analyzing program source using EXE [33], an older variant of the KLEE [24] program analysis platform. The idea is to mark memory locations—that is, taint [1] those locations—during symbolic execution and truncate exploration of paths that are algorithmically determined to produced effects that have been seen before. This dramatically reduces the state space since the symbolic execution engine is not revisiting duplicate states and repeatedly executing branch instructions that do not yield interesting results.

Godefroid [42] suggests he notion of state space exploration can be thought as the following: concurrent executions of branch instructions are partial orders where concurrent "independence" transitions should be left ordered. He asserts that transitions should be considered independent when the order of executions is irrelevant. Based on that idea, he uses this notion to determine possible "valid" dependency relations for branch executions of a given Labeled Formal Concurrent System (LFCS). Following that, he uses the logic that if a state is reachable by any sequence of transitions in a trace, it is sufficient to only explore one sequence in that trace, thus reducing the amount of state space required to visit during path exploration.

Godefroid and Khurshid [48] implemented a genetic algorithm framework to exploit heuristics for guiding search in the state space of concurrent reactive systems to find errors including deadlocks and assertion violations. The reasoning behind this is that a genetic algorithm will exploit heuristics that simulate natural-evolution processes like selection and mutation. Mapping this back to state space exploration, the algorithm measures the *fitness* of a path and determine whether that path is fit enough for exploration. Otherwise, a path deemed to be "fitter" for survival (i.e., net more interesting results) will be explored within a selection of branch instructions. This approach appears to outperform random and systematic searches when exploring large state spaces.

Li [49] proposed a context-sensitive relevancy analysis algorithm that uses weighted pushdown model checking to derive memory locations in a program where symbolic values can be inputted. The resulting output information is then utilized by a code instrumenter to transform relevant segments of a Java program with symbolic constructs. This creates a more precise path of execution by only executing branch instructions that appear to have more contextual relevancy to the function of the program.

Godefroid, Holzmann, and Pirottin [50] discussed the problem of storing states in the face of state explosion by describing states derived from branch executions that lead to the same state in subsequent state transitions. By determining those redundant states, and consequently determining that the most reachable states only get visited once during state exploration, the concept of state-space caching during program analysis can become a more viable option to reduce state space exploration and speed up analysis.

Despite all the claims made by academic researchers, Pelanek [51] reported in 2009 that many research papers make unjustified claims about respective techniques that attempt to manage the state explosion problem. Pelanek's complaint is that the research does not consider realistic evalution scenarios and includes poor experimental standards. As a result, this limits the practical application of the results.

CHAPTER 3: Symbolic Analysis Framework

When applying program analysis techniques, the primary technical challenge is approaching the analysis of a compiled binary itself and working with the machine code that makes up the binary. It is logical to assume that a typical analyst does not have access to source code for many of the Commercial-Off-the-Shelf (COTS) programs that are run on a computer. For example, a typical analyst would not have access to Microsoft's source code for the Windows OS or for Microsoft Word. If this analyst were astute enough to try to analyze the Word program binary code, it would quickly become clear to the user that the sheer volume of machine code to sift through would be overwhelming. Malware authors do not simply provide source code to the programs that are being distributed. As alluded to earlier in Chapter 2, malware authors frequently attempt to create countermeasures to program analysis by applying various anti-analysis techniques to raise the cost of time and resources for an analyst or analysis engine to correctly ascertain the behavior and function of the malware binary.

To that end, a symbolic analysis framework offers a logical method for analyzing machine code taken from a binary. Machine code is what the CPU fetches, decodes, and executes, therefore the logical conclusion is machine code will provide the ground truth for tests, evaluations, and applications. The following chapter is dedicated to providing some insight into the architecture of the framework and discuss the goals of the framework.

The architecture for the framework, is a modular and tightly integrated analysis suite. This suite consists of four core pieces:

1. The Core Emulation/Analysis Engine
2. The Algebra Engine
3. The Taint Engine
4. The SAT Solver

Figure 3 shows a notional diagram of the information flow between the framework elements.

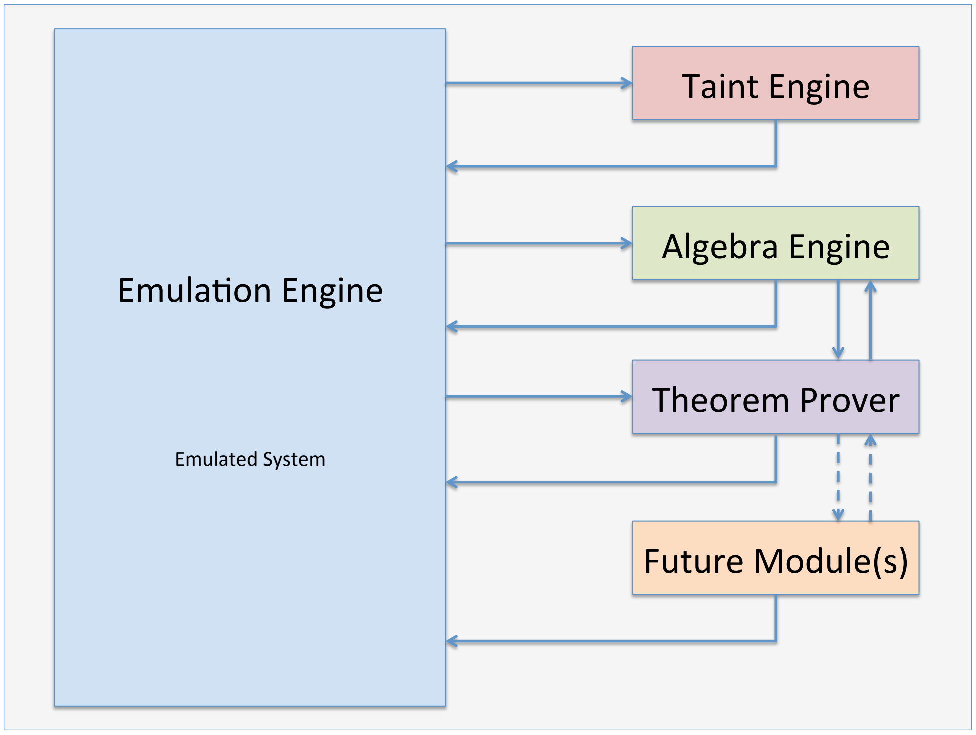


Figure 3.1: A Notional Diagram of Analysis Framework

3.1 Emulation Engine

The emulation engine of is the brain of the framework. It drives all other functionality and provides the information for the other framework elements to function. Functionally speaking, the emulation engine performs the program-wide symbolic execution, which allows granular monitoring and instrumentation.

The symbolic execution engine is considered *partial*-system because a user space environment exists for binary code to be executed. This includes the provision of a minimal set of OS services, virtual memory address space, system and application libraries, and even actual applications, allowing us to observe behavior from the application-layer down to the actual machine code being executed. There are several motivating factors that drive this approach:

- Analyzing software, particularly malware, requires introspection at the machine code and instruction level to discern interesting behavior.
- Like COTS virtual machine software, a platform that allows partial-system emulation offers realism for sandboxing during an analysis. In general, virtual environments can be

viewed as native operating environments to garner precise and accurate observations[2].

- Like COTS virtual machine software, there is generally good isolation between the analysis environment (sandbox), and the host machine, providing a safer environment to conduct analyses.
- An environment that allows instrumentation of the binary under test provides an environment conducive to isolating the analysis tools from the binary code under test to guard against interference during analysis or bad results.
- Analyzing binary code, for our purposes, requires software-level introspection to derive accurate and precise results. We need to be able to observe application-level interactions while concurrently observing instruction cycles as the program executes.
- We also require an ability to mark, or *taint* data—specifically, user-supplied data—and observe how it propagates through an application's state space.

The emulation engine acts as a virtual machine that has its own CPU, and its own allocated memory. When a 32-bit program is run with the engine, a 4GB virtual address space is mapped for a binary code under test. Associated libraries are mapped and monitored, which is critical for keeping track of how the control flow of a program progresses as the program state is explored.

The approach for our symbolic execution is *just-in-time* execution, where there is no apriori knowledge of the program before execution, and calculations are done as demanded. This idea is taken from the business concept where objectives are met through intelligently signaling different points of the workflow process to indicate when an event should occur [52]. The motivation behind this is that we endeavor to create an accurate depiction of how a program would execute on a system. The omniscient view requires apriori knowledge which, in most cases for a given framework, are derived from a disassembly. The problem with disassembly representations were mentioned in Chapter 2; Indeed, the primary concern is the accuracy of the disassembly output. If a program has anti-analysis techniques applied including anti-disassembly techniques, then the disassembly output may not be useful. Given this, we opted to eliminate the dependency on any type of omniscient view of a program for the time being.

During a program's symbolic execution, each instruction cycle is interpreted by the engine's CPU and the corresponding operation is performed against the current program state. The fetch-decode-execute cycle is performed by retrieving a program instruction, interpreting the action(s) required by decoding, and then executing the interpreted action(s). Within this process, the

[2]This assumes the binary code under test does not have anti-VM techniques implemented.

symbolic execution engine will signal other modules to perform analysis operations during each instruction cycle.

3.2 Algebra Engine

The name of the engine is taken from the fact that we are attempting to mathematically compute invariants of binary code under test by calculating the pre- and postconditions of a basic block of instructions. To summarily provide context, a basic block is a sequence of instructions with a single entry point and a single exit point. This means that the first instruction of the basic block denotes the start of the sequence of instructions, there is no jump instruction anywhere into or out of the sequence, and the last instruction of the block denotes the end of the sequence of instructions. When a basic block executes and the first instruction is executed, the rest of the instructions in the sequence must execute exactly once, in the exact order of the sequence.

We base the foundation of the algebra engine on the work of Reinbacher and Jörg [53]. Our goal is to build upon the foundations of their paper to accomplish the following objectives:

- Simplify re-calculation for redundant instructions.
- Allow the pausing and resuming of analysis. If a given basic block is already has pre- and postconditions calculated for a given set of inputs, then re-calculating over identical inputs is unnecessary.
- Assist with abstracting up for analysts to see the beginning and end results for a given basic block. This helps avoid “getting lost in the weeds”.
- Help with SAT-based operations to maintain soundness.

The algebra engine will run as a instrumentation tool that is signaled by the execution engine. Using logic built into the emulation engine, it will perform control flow and data flow analysis. Generated results from both the control flow and data flow analyses will be fed to the algebra engine where state will be recorded for each instruction that is retrieved.

Further discussion and exploration of the algebra engine will be discussed in Chapter 4.

3.3 Taint Engine

The taint engine is the framework module that will track and mark input that appears to come from a user. The reasoning behind taint analysis is that any input that comes from user-supplied data and modifies, or has the potential to modify, the flow of execution poses a risk to the binary that is being executed. For instance, a basic stack-based buffer overflow can occur if user-supplied data is able to write past the end of an allocated data buffer, resulting in the overwrite of a function return address. This can lead to the execution of user-supplied code which can lead to major security risks to the underlying operating system.

Common vulnerabilities such as stack-based buffer overflows, heap overflows, and format string vulnerabilities allow attackers to overwrite critical values in a program such as return addresses or function pointers that should not be derived from user input [54]. Based on such observations, dynamic taint analysis has gained popularity as it allows the tracking of user input and other untrusted data as it flows through a program at runtime and determines whether untrusted data is being used in an unsafe manner [1, 55].

Our taint engine aims to follow the basic taint workflows many other taint analysis frameworks contain:

Taint Tagging The process of defining what sources are untrusted and what should be tagged as tainted. This is determined before binary code execution and is usually associated with registers and program memory. These registers and memory space are initialized as untainted to begin with.

Taint Propagation The process of defining what computations allow the spreading of taint tags and how they propagate through the control flow of a binary code under execution. As an example, assignment operations such as `mov`, `add`, `sub`, `or`, etc. are x86 instructions that might help propagate tainted data, though there are edge cases where tainted data can be overwritten by untainted data. For example, `xor eax, eax` would result in tainted data residing in register `eax` to be zeroed out. Taint propagation runs concurrently with taint checking as each instruction will go through a taint propagation and taint check process.

Taint Checking The process where tainted data is being checked to see if it is being misused to alter program control by identifying "critical" computations or operation. Taint checking and taint propagation run concurrently since each instruction needs to be checked to see if it is unsafe. If an instruction is identified to be unsafe and is about to be executed, the taint status of all operands, registers, and memory are checked. That is, the *state* at that instruction is checked for taint. For example, an unconditional `jmp` instruction may be identified as a critical instruction.

Most taint methods simply track whether a bit in memory or in the general registers is tainted and provide no further information regarding where tainted data comes from. Our method aims to more precisely monitor the source of tainted data, rigorously specify what x86 instructions should propagate taint, and what instructions need to be monitored for misuse so that we can clearly delineate what values need to be tracked in order to characterize a potential vulnerability.

3.4 The Theorem Prover

The theorem prover is a module that takes path constraints that are derived from the pre- and post-conditions that the algebra engine generates and checks to see if path reachability is possible for a given set of inputs. Essentially, path constraints are a formula of Boolean variables that are calculated to see if it evaluates to *TRUE* (or *SAT*, short for *satisfied*). If the formula is not satisfiable, then the evaluation returns *FALSE* (or *UNSAT*, which stands for *unsatisfied*). The goal here is to determine whether a particular path taken in a program's execution control flow is actually reachable or not. The results are then provided back to the rest of the framework to derive other information that will further the analysis of binary code under test. One primary question surrounding the reachability of a path concerns the steps taken during symbolic execution to reach that path and what data was supplied during execution to get there. If there is potential for tainted (user-supplied) data to be provided to reach a path of interest, then it is possible to repeatedly generate the path conditions that lead to interesting actions of binary code under test. Current theorem proving tools utilize Boolean Satisfiability (SAT) decision procedures or Satisfiability Modulo Theories (SMT) decision procedures [34, 43, 56].

Theorem proving can also be understood as a decision problem. The Boolean formula is inputted into the solver which uses an algorithm to determine a "Yes" or "No". Symbolic execution is complemented by this approach to solving a decision problem from a formal system given how closely related they are. These algorithms, or *decision prcedures* [57, 58], allow for practical applications in the case of program verification. Most theorem prover decision procedures are based on the DPLL algorithm, developed by Davis, Putnam, Logemann, and Loveland [29, 59]. Contemporary decision procedures take advantage of SMT techniques, since SMT builds on SAT solving techniques and offers a more expressive modeling language using first-order logic [56, 58].

In the context of our framework, the theorem prover will be signaled by the algebra engine to solve a path condition for a basic block where the pre- and postconditions have been calculated by the algebra engine and translated into a model for the theorem prover to work with. Results will be recorded for the symbolic engine to bias itself towards paths that have yet to be explored.

Our goal is to provide soundness to the path constraints we calculate in order to determine reachability and derive a set of path conditions we can use to reconstruct concrete inputs to reach interesting paths of execution and observe program behavior.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Reasoning Over Native x86

Reasoning over the Intel native x86 instruction set presents a significant challenge but presents some clear advantages. First, we do not have an intermediate language (IL), so we do not deal with the potential for losing information in translation. Intermediate languages are certainly useful as compiler front-ends use them when analyzing a piece of source code that gets passed to the compiler back-end to generate machine code for a given target platform [26]. Traditional approaches to symbolic analysis of binary code leverage aspects of the compilation process by using facets of it to semantically approximate the reasoning of the binary code. In other words, intermediate languages assist in the symbolic analysis process to approximate the decompilation process of the binary code as close as possible to the original source code.

Indeed, researchers in the field gravitate toward ILs because they reduce semantic complexity during analysis. Frameworks such as LLVM [22] have gained tremendous popularity because of their usefulness in the compilation process. ILs have several applications. One example is generating an intermediate representation (IR) of a piece of source code and performing code transformations. This is a critical highlight when discussing interoperability since this means compilers can be used across different OS platforms. By using an intermediate language that is common among different compiler platforms, a piece of source code may have the semantics retained and honored when the back-end compiler produces the machine code for a given target platform.

As mentioned earlier, the advantages of using an IL are accompanied by some disadvantages. The depth level of an IL is critical when dealing with machine dependence. For example, high level ILs are used almost entirely in the earliest stages of the compilation process, or in preprocessors before compilation [21]. In some ways, a high level IL can be thought of as a pseudocode representation of source code before the actual source program is written [21]. A medium level IL is generally designed to reflect the range of features in a set of source languages, but in a language-independent manner. They are designed to be good bases for generating efficient machine code for at least one architecture [21]. With languages such as C/C++, medium-level ILs are the de facto standard level since they are appropriate for most of the optimizations done in compilers, such as common-subexpression elimination, code motion, and algebraic simplification [21]. Low-level ILs often provide a nearly one-to-one mapping to target machine instructions and can generally be considered architecture-dependent. The IRs derived from low-level ILs allow maximal optimization to be performed on the intermediate code and in the final stages of compilation to either expand intermediate-code instructions into code sequences or to combine related ones into more powerful instructions [21]. There exists multi-level intermediate languages that consider a piece of source code and the need to represent it at different levels through the compilation process in order to best transform it into platform-dependent machine code [21]. Regardless of level, intermediate languages are still approximating the semantics of source code to binary code. This can lead to fidelity loss in the precision of expression when trying to simplify the instruction semantics of an architecture such as the Intel ISA.

The expressiveness of an IL is dependent on how extensive its syntax is to express every instruction. It is generally thought that simplifying seemingly identical instructions makes sense. However, when analyzing binary code such as the Intel instruction set, it is not possible since operations and operands can vary in length, from one to twelve or more bytes. Using an IL to simplify or encompass all `mov` instructions, for instance, may help abstract away some of the need for precision but, using a catch-all instruction may lead to an over-approximation or improper approximation of what may be a completely different instruction [53]. The precision of any IL is subject to this problem. Our goal is to maintain the highest level of fidelity possible when looking at an instruction by maintaining a contextual look at native x86 and do away with an IL.

The types of transformations that can be done using an intermediate language are not as important for binary code analysis as the expressiveness or the depth level of the IL, but it is important to understand how compiler ILs perform transformations to understand why they can dilute the fidelity of binary code. IL transformation algorithms consider the depth level and the expression syntax that are part of the language design. The transformation described generally tend toward the simplification of the source as it is being translated into machine code or the optimization of the aforementioned code by reducing the complexity. For example, a level of optimization may determine that converting certain arithmetic loop operations may be more efficient by either converting the loop into a long sequence thereby eliminating the loop, converting the arithmetic operation itself into another operation, or both. By the time the machine code has been generated, much of the code may not resemble the actual source code. If we were to analyze this same optimized binary code, and translate it back into an intermediate language fit for analysis, the semantics of this code would further have to be simplified, which could lead to a dilution of fidelity. As it is, binary code analysis is difficult enough without having to work with lossy results. To that end, we maintain that an intermediate language is not a desirable “extra” step we want to take when performing analysis something like malware, for example.

4.1 Developing Reasoning Over x86 Architecture

To begin developing the algebra we need to reason over the x86 instruction set, it needs to be understood that there are different levels of *program context* we are trying to develop. Program context is the representation of our state at different levels during execution and is the container for all of our analysis information. Taking inspiration from Fahringer and Scholz [60], we need to define what our *program context* is and how the different levels of context tie together. We want to develop context at three levels:

1. At each instruction
2. At each basic block
3. Through the entire program during execution

We now need to understand how path conditions propagate in accordance with the x86 architecture and become path constraints. We understand that path constraint derivation will largely be a product of our program context but will also require understanding the behavior of key assembly-level operations and the mnemonics of the Intel ISA.

In this thesis, we limit our discussion to the following instruction categories:

- Branching
- Flag Setting
- Arithmetic
- Logic
- Data Transfer

These instruction categories represent a base of common instructions that are regularly encountered for OS and library call operations.

Instruction categories that are considered outside the scope of this thesis include the following:

- Floating Point
- MMX
- Streaming Single Instruction, Multiple Data (SIMD) Extensions (SSE) [61]

4.2 Establishing Semantics

Based on work by Reinbacher and Tseitin [53, 62], we begin with establishing the semantics that will be used to derive our program context. Following this, we will apply these semantics to a single line instruction, then lift these techniques up to a basic block level, and then finally use the pre- and postconditions we have generated to describe the state of a program at a given point in the program.

We have several ingredients that we will use to derive our instruction-level program context *S*, based on [60]. We can define the program state as the following:

- The 32-bit address space a running process is allocated. This includes:
  - Addressable memory locations
- The set of registers:
  - `eax`
  - `ebx`
  - `ecx`
  - `edx`
  - `edi`
  - `esi`
  - `esp`
  - `ebp`
  - `eip`
- The set of segment registers that hold 16-bit selectors:
  - `cs`
  - `ds`
  - `ss`
  - `es`
  - `fs`
  - `gs`
- The 32-bit EFLAGS register (holds status flags):
  - `OF` (bit 11)—Overflow Flag
  - `SF` (bit 7)—Sign Flag
  - `ZF` (bit 6)—Zero Flag
  - `AF` (bit 4)—Adjust Flag

  - `PC` (bit 2)—Parity Flag
  - `CF` (bit 0)—Carry Flag
- Static Single Assignment (SSA) Form Variables [21,63]

The SSA variables have the property that each variable is assigned exactly once during the analysis of a program. This provides the advantage of being able to distinctly verify every variable that is created during analysis. Since there is no variable reuse, it is possible to determine which variables were used during a back or forward trace of an execution path [21,53].

Next, we will formulate the algebra that mathematically describes instruction state, basic block state, and program state.

4.2.1 Instruction Encoding

With the established semantics, we can proceed to transform each instruction, based on work by Reinbacher and Tseitin [53,58,62].

Definition: Instruction Encoding

A single *instruction encoding* is described as a mnemonic M followed by its one or two operands.

$$[[\,\mathbf{M}\,\beta[,\alpha]\,]] \;:=\; (S_{mem\_addr} \setminus \{\beta\}) \cup \{\beta'\} \;=\; \bigwedge_{i=0}^{n-1} \text{operation}_{mem\_addr} \tag{4.1}$$

where:

- S_{mem_addr} is the set that describes the program context upon entrance of the instruction.
- α and β are operands. β can be considered the *destination* and α can be considered the *source* to stay consistent with Intel-style semantics.
- $(S_{mem\_addr} \setminus \{\beta\}) \cup \{\beta'\}$ is the set difference subtracting β from the set and adding β' to the set. This describes the derived program state as a result of the instruction's operation.
- $\bigwedge$ represents the conjunction of Tseitin-encoded clauses.
- i is the bit counter.
- n represents the number of bits for the operand.
- $operation_{mem\_addr}$ is the instruction operation that is derived from the Intel ISA.

Figure 4.1: Instruction Encoding Definition

As an example, if we wanted to express an *add* operation over arbitrary values a and b at some arbitrary memory address, say, `0xbffffdc0`, we would express it as follows:

$$[[\,\mathbf{ADD\;a},\mathbf{b}\,]] \;:=\; (S_{0xbffffdc0} \setminus \{\mathbf{a}\}) \cup \{\mathbf{a}'\} \;=\bigwedge_{i=0}^{n-1}\left(\mathbf{a}' \;\leftrightarrow\; a+b\right) \tag{4.2}$$

where *a'* is the result upon exit of the completed instruction.

Single operand expressions, such as an *inc* operation, over an arbitrary value *c* at some memory location (we will re-use `0xbffffdc0`) can be expressed as follows:

$$[[\,\mathbf{INC\;c}\,]] \;:=\; (S_{0xbffffdc0} \setminus \{\mathbf{c}\}) \cup \{\mathbf{c}'\} \;=\bigwedge_{i=0}^{n-1}\left(\mathbf{c}' \;\leftrightarrow\; c+1\right) \tag{4.3}$$

For bitwise operations, such as an *xor* operation at memory location `bffffdc0`, we would express it as follows:

$$[[\,\mathbf{XOR\;d},\mathbf{e}\,]] := (S_{0xbffffdc0} \setminus \{\mathbf{d}\}) \cup \{\mathbf{d}'\} \;=\bigwedge_{i=0}^{n-1}\left(\mathbf{d}'[\mathbf{i}] \;\leftrightarrow\; d[i] \,\oplus\, e[i]\right) \tag{4.4}$$

Similar instruction encodings can be derived for the entire Intel ISA. As each instruction is encoded in this set and bitwise fashion, we can extend it to the basic block level by constructing a basic block formula ϕ that is a conjunction of all the instruction contexts within that given block. This can be generically represented as follows:

4.2.2 Basic Block Encoding

Consider a generic basic block of n instructions. We can say that the resulting output from an instruction I_i is state S_{i+1} that will be used as input to instruction I_{i+1}. Figure 4.2 describes this calculation process:

Equations (4.5) and (4.6) describe how to mathematically describe basic block state with respect to the instructions contained in a basic block.

This can be elaborated further by showing the set operations between each instruction as shown in Figure 4.4.

$$\begin{aligned}
\mathbf{I_0} &= [[\ \mathbf{M}_0\ \beta_0, \alpha_0\ ]] = \mathbf{S_1} \\
\mathbf{I_1} &= [[\ \mathbf{M}_1\ \beta_1, \alpha_1\ ]] = \mathbf{S_2} \\
\mathbf{I_2} &= [[\ \mathbf{M}_2\ \beta_2, \alpha_2\ ]] = \mathbf{S_3} \\
&\vdots \\
\mathbf{I_{n-1}} &= [[\ \mathbf{M}_{n-1}\ \beta_{n-1}, \alpha_{n-1}\ ]] = \mathbf{S_n} = \psi_{bb}
\end{aligned}$$

Figure 4.2: Generic Basic Block of *n* Instructions

Definition: Basic Block Encoding

The resulting state derived by the n^{th} instruction operation provides the final state for the basic block indicated by S_{mem_addr} and can be written as the following sequence:

$$\psi_{\mathbf{bb}} = \bigwedge_{i=0}^{n-1} [[\ \mathbf{M}_i\ \beta_i, \alpha i\ ]] \tag{4.5}$$

where:

- $\bigwedge$ denotes a sequence of instruction that results in the final state computed from the n^{th} instruction, I_{n-1}.
- The computation to derive S_{i+1} is derived from Equation (4.1).

and can be re-written with the set operation from Equation (4.1) as follows:

$$\psi_{\mathbf{bb}} = \bigwedge_{i=0}^{n-1} (S_i \setminus \{\beta_{\mathbf{i}}\}) \cup \{\beta'_i\} \tag{4.6}$$

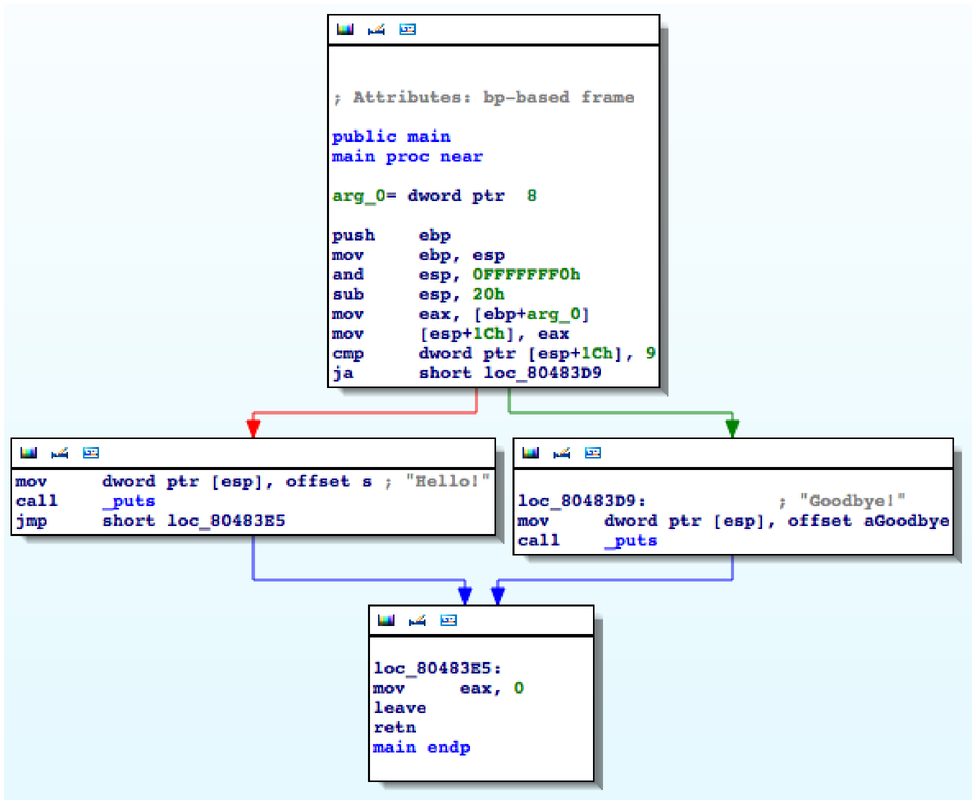
Figure 4.3: Basic Block Encoding Definition

$$\begin{aligned}
\mathbf{I_0} &= [[\ \mathbf{M}_0\ \beta_0, \alpha_0\ ]] := (S_0 \setminus \{\beta_0\}) \cup \{\beta'_0\} = \mathbf{S_1} \\
\mathbf{I_1} &= [[\ \mathbf{M}_1\ \beta_1, \alpha_1\ ]] := (S_1 \setminus \{\beta_1\}) \cup \{\beta'_1\} = \mathbf{S_2} \\
\mathbf{I_2} &= [[\ \mathbf{M}_2\ \beta_2, \alpha_2\ ]] := (S_2 \setminus \{\beta_2\}) \cup \{\beta'_2\} = \mathbf{S_3} \\
&\vdots \\
\mathbf{I_{n-1}} &= [[\ \mathbf{M}_{n-1}\ \beta_{n-1}, \alpha_{n-1}\ ]] := (S_{n-1} \setminus \{\beta_{n-1}\}) \cup \{\beta'_{n-1}\} = \mathbf{S_n} = \psi_{bb}
\end{aligned}$$

Figure 4.4: Basic Block of Instructions with Set Difference Break Out

4.2.3 Program Path Encoding

Raising our semantics up to the program level presents issues we must now strongly consider. One of the primary concerns is how to maintain state at instructions that create branching. When a program branches, there are at least two paths that the program can take during execution. For example, consider Figure 4.5



```
; Attributes: bp-based frame

public main
main proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
mov     eax, [ebp+arg_0]
mov     [esp+1Ch], eax
cmp     dword ptr [esp+1Ch], 9
ja      short loc_80483D9
```

```
mov     dword ptr [esp], offset s ; "Hello!"
call    _puts
jmp     short loc_80483E5
```

```
loc_80483D9:                 ; "Goodbye!"
mov     dword ptr [esp], offset aGoodbye
call    _puts
```

```
loc_80483E5:
mov     eax, 0
leave
retn
main endp
```

Figure 4.5: A Simple C Program with Branching (Taken from IDA Pro)

For a particular path p_0, the values contained in our program state will change and be different from the state that they will be in p_1. Using the example above, we can see that in order for the program to print out "Hello!", the user-supplied input must contain between 0 and 8 arguments, since the name of the program being executed is also part of the argument vector, which adds 1 argument. The program outputs "Goodbye" for any other argument count (anything that is not between 0 and 9 total arguments).

Up until the state reaches the instruction `0x080483c9: ja short lo_80483D9`, our basic block state—and our program state up to this point of 'execution—will be a result from the output derived from not the compare instruction, `0x080483c4: cmp dword ptr [esp+1Ch], 9`, but rather the data transfer instruction, `0x080483c0: mov [esp+1Ch], eax`. Once the `ja` instruction has executed, then we can determine that there is a conditional branch which can be traced back to the `cmp` instruction which compared the argument count with the value "9". This means that there are two distinct paths that can be taken, one we will call p_0, which corresponds to the path that reaches the "Hello!" statement. The second path we call p_1, which corresponds to the path that reaches the "Goodbye!" statement.

To maintain simplicity in this example, we will only look at a small set of values within our state that we will call S_5 to correspond to line 6 of the instructions in Figure 4.5, `mov [esp+1Ch], 9`. We can see that register `eax` is what contains the value of the argument count that is user-influenced. From here, we can see that the data transfer that takes place is the value from eax is copied to `esp+1Ch` which corresponds to a 28 byte displacement from where the stack pointer `esp` currently points. The value at that memory location is compared to see if it is greater than the value of 9. If it is, then it jumps to the basic block that would print "Goodbye!". If the value is less than 9, then it prints out "Hello!". We can deduce that the state differences are the following:

- For path p_0, the state value for `eax` must be between 0 and 9 in order for this path to be executed.
- For path p_1, the state value for `eax` must be anything that is not between 0 and 9 for this path to be executed.

In order to mathematically represent the contextual differences between the respective state for paths p_0 and p_1, we introduce the symbol ϕ which is a function of the sequence of traversed basic blocks, and therefore, all the instruction that were evaluated to reach their respective points of execution.

For the example above, our states prior to the conditional `ja` instruction would look like the following if *3* user-supplied arguments were supplied to the program for path p_0 and if *10* user-supplied arguments were supplied to the program for path path p_1[3][4]:

[3]Note 1: Since the other registers have not been used thus far, they are zeroed out.

[4]Note 2: The stack, base, and instruction pointer addresses illustrated here are not reflective of what they may be in our analysis framework.

$$
\begin{aligned}
\phi_{p_0} = \Bigg\{ & \{[ebp+0x8],[esp+0x1c]\}, \\
& \{eax = 0x4, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0, \\
& ebp = 0xbffffd6c, esp = 0xbffffd40, eip = 0x080483c9\}, \\
& \{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0, gs = 0x33\}, \\
& \{of = 0, sf = 1, zf = 0, af = 1, pf = 0, cf = 1\}, \{\emptyset\} \Bigg\}
\end{aligned}
$$

$$
\begin{aligned}
\phi_{p_1} = \Bigg\{ & \{[ebp+0x8],[esp+0x1c]\}, \\
& \{eax = 0xb, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0, \\
& ebp = 0xbffffd6c, esp = 0xbffffd40, eip = 0x080483c9\}, \\
& \{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0, gs = 0x33\}, \\
& \{of = 0, sf = 0, zf = 0, af = 0, pf = 0, cf = 0\}, \{\emptyset\} \Bigg\}
\end{aligned}
$$

As can be seen in the example, slight variations in the state can result in radically different outcomes with path execution.

In order to precisely describe the state mathematically, we see that we first need to consider the basic blocks that have been traversed. Second, we can see that we also need to evaluate each instruction contained in each basic block. Third, we need to keep track of the state change between each instruction. Our semantics sufficiently capture all of this information, allowing us to come up with a general equation that accurately describes this, as seen in Figure 4.6

Definition: Program Path Encoding

The overall state of a program can be computed given a particular path of execution p that is followed. This can be defined as the following:

$$\phi_p = \bigwedge_{j=0}^{m-1} \psi_{\mathbf{j}} \tag{4.7}$$

where:

- p is a path of execution that is being inspected.
- j is the block counter
- m is the number of blocks in path p
- ψ_j is computed from Equation (4.5).

If we expand ψ_i in Equation (4.7) using Equation (4.5) and Equation (4.6), we will have:

$$\begin{aligned}
\phi_p &= \bigwedge_{j=0}^{m-1} \psi_{\mathbf{j}} && (4.8)\\
&= \bigwedge_{j=0}^{m-1} \left(\bigwedge_{i=0}^{n-1} [[\, \mathbf{M}_i\ \beta_i, \alpha i\,]] \right)_j && (4.9)\\
&= \bigwedge_{j=0}^{m-1} \left(\bigwedge_{i=0}^{n-1} (S_i \setminus \{\beta_{\mathbf{i}}\}) \cup \{\beta'_i\} \right)_j && (4.10)
\end{aligned}$$

where both $\bigwedge$ respectively represent a sequence of computations that result in the program state up to the point of path p's execution.

Figure 4.6: Program Path Encoding Definition

4.3 Applying the Math: A Worked Example

We have established an algebra for mathematically describing the state of a code under execution at the instruction, basic block, and program level with respect to a path of execution. We can now demonstrate the use of it with an example.

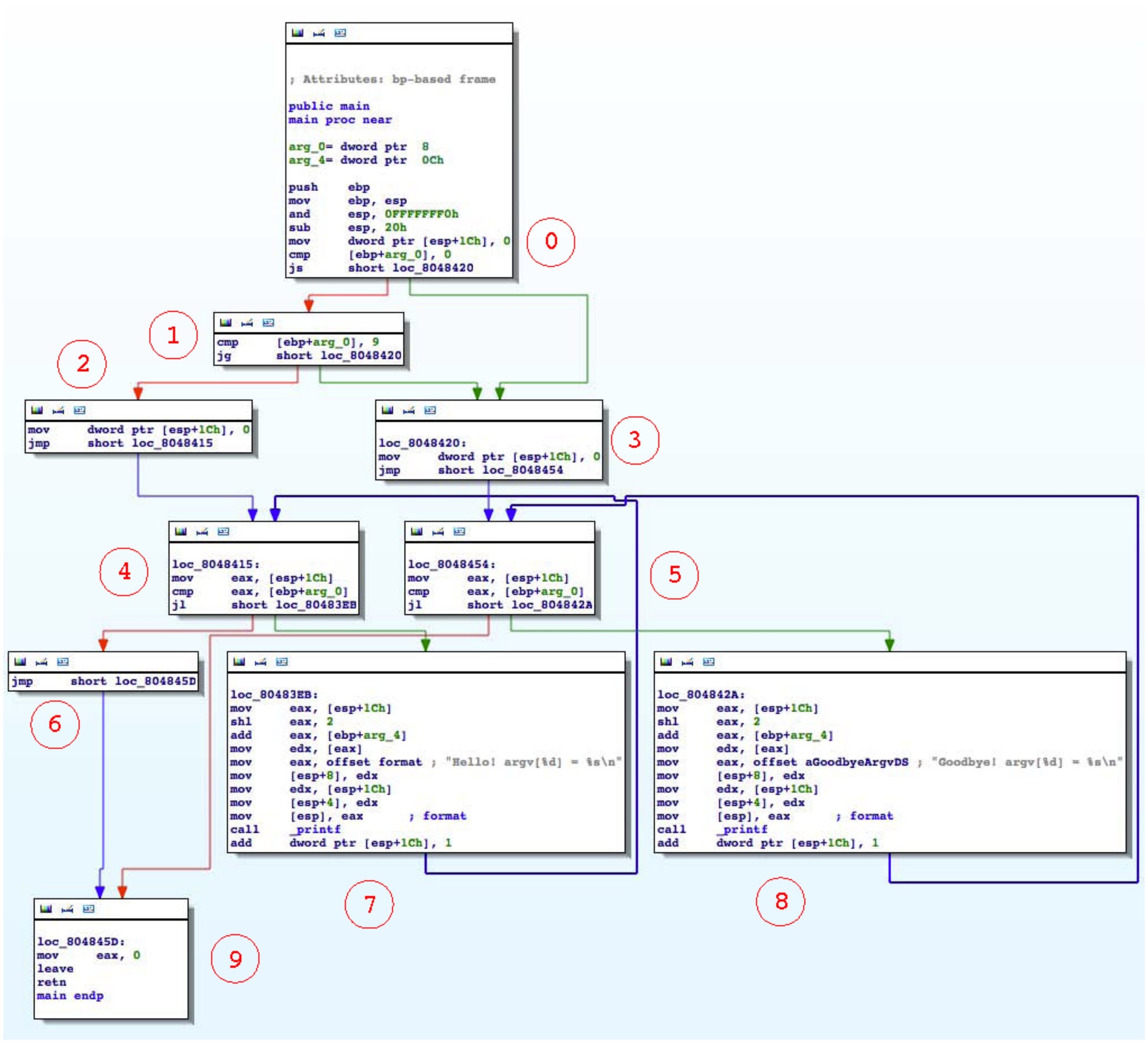
Expanding on our *simple-branch* example, we will analyze *loop-branch*, a program that will output "Hello!" or "Goodbye!", depending on the number of arguments given, just like simple-branch. Specifically, if the number of total arguments is between 0 and 9, then the program will output "Hello!". If the argument count is anything outside of that count, then it will output "Goodbye!". However, the additional function of the program is that it will also output all arguments contained in the argument vector. Figure 4.7 shows two examples of program output.

```
]# ./loop-branch a b c d e
Hello! argv[0] = ./loop-branch
Hello! argv[1] = a
Hello! argv[2] = b
Hello! argv[3] = c
Hello! argv[4] = d
Hello! argv[5] = e

]# ./loop-branch 1 2 3 4 5 6 7 8 9 10
Goodbye! argv[0] = ./loop-branch
Goodbye! argv[1] = 1
Goodbye! argv[2] = 2
Goodbye! argv[3] = 3
Goodbye! argv[4] = 4
Goodbye! argv[5] = 5
Goodbye! argv[6] = 6
Goodbye! argv[7] = 7
Goodbye! argv[8] = 8
Goodbye! argv[9] = 9
Goodbye! argv[10] = 10
```

Figure 4.7: *loop-branch* Output Example

To begin analysis, consider the CFG of *loop-branch* showing in Figure 4.8. The numbers in circles correspond to the assigned basic block number and basic block code that will be referenced during analysis.



```
; Attributes: bp-based frame

public main
main proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
mov     dword ptr [esp+1Ch], 0
cmp     [ebp+arg_0], 0
js      short loc_8048420
```

```
cmp     [ebp+arg_0], 9
jg      short loc_8048420
```

```
mov     dword ptr [esp+1Ch], 0
jmp     short loc_8048415
```

```
loc_8048420:
mov     dword ptr [esp+1Ch], 0
jmp     short loc_8048454
```

```
loc_8048415:
mov     eax, [esp+1Ch]
cmp     eax, [ebp+arg_0]
jl      short loc_80483EB
```

```
loc_8048454:
mov     eax, [esp+1Ch]
cmp     eax, [ebp+arg_0]
jl      short loc_804842A
```

```
jmp     short loc_804845D
```

```
loc_80483EB:
mov     eax, [esp+1Ch]
shl     eax, 2
add     eax, [ebp+arg_4]
mov     edx, [eax]
mov     eax, offset format ; "Hello! argv[%d] = %s\n"
mov     [esp+8], edx
mov     edx, [esp+1Ch]
mov     [esp+4], edx
mov     [esp], eax      ; format
call    _printf
add     dword ptr [esp+1Ch], 1
```

```
loc_804842A:
mov     eax, [esp+1Ch]
shl     eax, 2
add     eax, [ebp+arg_4]
mov     edx, [eax]
mov     eax, offset aGoodbyeArgvDS ; "Goodbye! argv[%d] = %s\n"
mov     [esp+8], edx
mov     edx, [esp+1Ch]
mov     [esp+4], edx
mov     [esp], eax      ; format
call    _printf
add     dword ptr [esp+1Ch], 1
```

```
loc_804845D:
mov     eax, 0
leave
retn
main endp
```

Figure 4.8: CFG of Program *loop-branch* (Taken from IDA Pro)

We can consider an initial state that the program's memory space will take on at the program's "main" function. For the sake of brevity, we will analyze one full basic block of instructiond starting at Basic Block 0 (Figure 4.9) and gradually raise up the mathematical analysis. In implementation, our analysis framework will analyze each instruction. Also, paths that may not be reachable will be briefly discussed but not analyzed in-depth.

4.3.1 Mathematical Analysis of *loop-branch*

Beginning with Basic Block 0 (Figure 4.9), we have an initial state to consider. With some stack offset that occurs when the process is "loaded" into memory, as well the pre-loading activities that occurred for our program to get to `main`, let's assume `esp` is pointing to address `0xbffff7f8` and `eip` is pointing at `11. 0x080483c4: push ebp`, which corresponds to line 11 of Basic Block 0..

```
1  .text:080483C4 ; =============== S U B R O U T I N E =======================================
2  .text:080483C4
3  .text:080483C4 ; Attributes: bp-based frame
4  .text:080483C4
5  .text:080483C4                 public main
6  .text:080483C4 main            proc near               ; DATA XREF: _start+17o
7  .text:080483C4
8  .text:080483C4 arg_0           = dword ptr  8
9  .text:080483C4 arg_4           = dword ptr  0Ch
10 .text:080483C4
11 .text:080483C4                 push    ebp
12 .text:080483C5                 mov     ebp, esp
13 .text:080483C7                 and     esp, 0FFFFFFF0h
14 .text:080483CA                 sub     esp, 20h
15 .text:080483CD                 mov     dword ptr [esp+1Ch], 0
16 .text:080483D5                 cmp     [ebp+arg_0], 0
17 .text:080483D9                 js      short loc_8048420
```

Figure 4.9: *loop-branch* Basic Block 0 (Taken from IDA Pro)

Our initial state, with our assumptions, will look like the Figure 4.10.

$$\phi_{initial} = \Bigg\{ \{\emptyset\}, \{eax = 0, ebx = 0, ecx = 0, edx = 0x8, edi = 0, esi = 0,$$
$$ebp = 0xbffff748, esp = 0xbffff6cc, eip = 0x080483c4\},$$
$$\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},$$
$$\{of = 0, sf = 0, zf = 1, af = 0, pf = 1, cf = 0\}, \{\emptyset\} \Bigg\}$$

Figure 4.10: Initial *loop-branch* Program State

Here, $\phi_{initial}$ means we are at the initial state of the program, which can be lowered down to the basic block, and therefore, the instruction level, where $\mathbf{S}_i$ contains same initial state values.

From here, if we execute `push ebp`, then we would observe the following as seen in Figure 4.11. Note that `push ebp` is encoded in a manner that affects `esp`, since the `push` operand decrements the stack pointer and then stores the source operand on top of the stack [61]:

$$\mathbf{I}_0 = [[\ \mathbf{PUSH\ ebp}\ ]] \ := \ (S_{0x080483c4} \setminus \{\mathbf{esp}\}) \cup \{\mathbf{esp'}\} \ = \ S_{0x080483c5}$$
$$= \bigwedge_{i=0}^{n-1} \left(\left[\mathbf{esp'} \ \leftrightarrow \ (esp' = esp - 4) \ \wedge \ (temp = ebp) \ \wedge \ ([esp'] = temp) \right] \right)$$

Figure 4.11: `push ebp` Instruction Encoding

Following this instruction, we can encode the succeeding instruction in a similar fashion. Figure 4.12 shows the instruction encodings for Basic Block 0.

After finishing the analysis of Basic Block 0, we can see that the state at S_{CMP} is interesting as the state here is what is compared in the next instruction and determines the branching. From our vantage point, we already know that we are approaching a branch in our analysis of the code under test. However, in implementation, our engine will not necessarily have our omniscient view since it is performing just-in-time symbolic execution. Therefore, we introduce a state called $S_{!BRANCH!}$ so that the engine will know that it has reached a branching instruction. All conditional and unconditional jumps will automatically receive this state so that it knows it has reached an instruction that produces a branch.

With the `cmp` instruction, we can determine the values that determine the branch condition. In Basic Block 0 (Figure 4.9), we can see that the two values being compared are the contents of a memory address and the value 0. Based on the operation of the `cmp` instruction, we know that the operation will set any of the EFLAGS based on the result. The `js` instruction following the `cmp` instruction indicates that the sign flag (SF) is checked to see if it is set to value 1, indicating that the result from the `cmp` instruction was in a negative number. Therefore, we know that if the result is negative, then a conditional jump will be taken to Basic Block 3 (Figure 4.14). However, if the result is not negative, then execution will continue to Basic Block 1 (Figure 4.13).

$$\mathbf{I}_0 = [[\ \mathbf{ebp}\ ]] \ := \ (S_{0x080483c4} \setminus \{\mathbf{esp}\}) \cup \{\mathbf{esp}'\} \ = \ S_{0x080483c5}$$
$$= \bigwedge_{i=0}^{n-1} \left(\mathbf{esp}' \ \leftrightarrow \ (esp' = esp - 4) \ \wedge \ (temp1 = ebp) \ \wedge \ ([esp'] = temp1) \right)$$

$$\mathbf{I}_1 = [[\ \mathbf{MOV\ ebp, esp}\ ]] \ := \ (S_{0x080483c5} \setminus \{\mathbf{ebp}\}) \cup \{\mathbf{ebp}'\} \ = \ S_{0x080483c7}$$
$$= \bigwedge_{i=0}^{n-1} \left(\mathbf{ebp}' \ \leftrightarrow \ ebp' = esp \right)$$

$$\mathbf{I}_2 = [[\ \mathbf{AND\ esp, 0xfffffff0}\ ]] \ := \ (S_{0x080483c7} \setminus \{\mathbf{esp}\}) \cup \{\mathbf{esp}'\} \ = \ S_{0x080483ca}$$
$$= \bigwedge_{i=0}^{n-1} \left(\mathbf{esp}' \ \leftrightarrow \ \left[(temp2 = 0xfffffff0]) \ \wedge \ (esp'[i] \ \& \ temp2[i]) \right] \right)$$

$$\mathbf{I}_3 = [[\ \mathbf{SUB\ esp, 0x20}\ ]] := \ (S_{0x080483ca} \setminus \{\mathbf{esp}\}) \cup \{\mathbf{esp}'\} \ = \ S_{0x080483cd}$$
$$= \bigwedge_{i=0}^{n-1} \left(\mathbf{esp}' \ \leftrightarrow \ esp - 0x20 \right)$$

$$\mathbf{I}_4 = [[\ \mathbf{MOV\ dword\ ptr\ [esp + 0x1C], 0}\ ]]$$
$$:= \ (S_{0x080483cd} \setminus \{\mathbf{dword\ ptr\ [esp + 0x1C]}\}) \cup \{(\mathbf{dword\ ptr\ [esp + 0x1C]})'\} \ = \ S_{0x080483d5}$$
$$= \bigwedge_{i=0}^{n-1} \left((\mathbf{dword\ ptr\ [esp + 1C]})' \ \leftrightarrow \ (dword\ ptr\ [esp + 1C] = 0) \right)$$

$$\mathbf{I}_5 = [[\ \mathbf{CMP\ [ebp + 8], 0}\ ]] \ := \ (S_{0x080483d5} \setminus \{\mathbf{temp3}\}) \cup \{\mathbf{temp3}'\} \ = \ S_{0x080483d9}$$
$$= \bigwedge_{i=0}^{n-1} \left(\mathbf{temp3}' \ \leftrightarrow \ \left(([ebp + 8] - 0) \ \wedge \ (modify(EFLAGS)) \right) \right)$$

$$\mathbf{I}_6 = [[\ \mathbf{JS\ 0x08048420}\ ]] \ := \ (S_{0x080483d9} \setminus \{\mathbf{temp4}\}) \cup \{\mathbf{temp4}'\} \ = \ S_{!BRANCH!}$$
$$= \bigwedge_{i=0}^{n-1} \left(\mathbf{temp4}' \ \leftrightarrow \ \left((SF == 1) \wedge (temp4 = eip + 0x08048420) \wedge (eip = temp4) \right) \right)$$

Figure 4.12: Instruction Encodings for Basic Block 0

```
.text:080483DB                cmp     [ebp+arg_0], 9
.text:080483DF                jg      short loc_8048420
```

Figure 4.13: *loop-branch* Basic Block 1 (Taken from IDA Pro)

```
.text:08048420
.text:08048420 loc_8048420:                            ; CODE XREF: main+15j
.text:08048420                                         ; main+1Bj
.text:08048420                mov     dword ptr [esp+1Ch], 0
.text:08048428                jmp     short loc_8048454
```

Figure 4.14: *loop-branch* Basic Block 3 (Taken from IDA Pro)

Based on this analysis, we know what the two possible basic block states are to reach Basic Block 1 or Basic Block 3, and thus, the two possible program states as shown in Figure 4.15 and Figure 4.16.

$$
\begin{aligned}
\psi_{bb0}^{0} = \phi_{p_0}^{0} = \Big\{ & \{[ebp+0x8] \geq 0, [esp+0x1c] = 0\},\\
& \{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,\\
& ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x080483db\},\\
& \{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},\\
& \{of = 0, sf = 0, zf = 0, af = 0, pf = 0, cf = 0\},\\
& \{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,\\
& temp4 = eip + 0x080483db\}\Big\}
\end{aligned}
$$

Figure 4.15: State that causes Basic Block 0 to Branch to Basic Block 1

$$\psi_{bb0}^{1} = \phi_{p_0}^{1} = \Big\{ \{[ebp + 0x8] < 0, [esp + 0x1c] = 0\},$$
$$\{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,$$
$$ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x08048420\},$$
$$\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},$$
$$\{of = 0, sf = 1, zf = 0, af = 0, pf = 0, cf = 0\},$$
$$\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp + 8] - 0,$$
$$temp4 = eip + 0x08048420\} \Big\}$$

Figure 4.16: State that causes Basic Block 0 to Branch to Basic Block 3

Continuing the analysis at the basic block level, we will see that Basic Block 3 (Figure 4.14) is a basic block that acts as a lead-in for Basic Block 5 (Figure 4.17), a basic block that either branches to the end of the program for exiting, or branches to Basic Block 8 (Figure 4.28). This is important to note as Basic Block 5 and Basic Block 4 (which has not yet been discussed) are both basic blocks that act as loop counters for outputting the argument vectors that are user-supplied at the start of the program. In order to reach Basic Block 3, and subsequently Basic Blocks 5 and 8, the sign flag (SF) needs to be set in order to perform the jump from Basic Block 0 to Basic Block 3. This means that there needs to be either a negative number of inputs or enough inputs that cause an integer overflow of the argument count. Whether this is actually possible or not is beyond the scope of this thesis, but we will discuss executing down a feasible path that outputs “Hello!” with an argument vector that is a count between 0 and 9.

```
.text:08048454 loc_8048454:                               ; CODE XREF: main+64j
.text:08048454                 mov     eax, [esp+1Ch]
.text:08048458                 cmp     eax, [ebp+arg_0]
.text:0804845B                 jl      short loc_804842A
.text:0804845D
```

Figure 4.17: *loop-branch* Basic Block 5 (Taken from IDA Pro)

Going back to the branching at the exit of Basic Block 0, we've noted that Basic Block 0 can branch to Basic Block 1 if the output is positive since S_{CMP} would subsequently have the sign flag set to `SF == 0`. Analyzing this path of execution will result in ψ_{bb1} after the `line 1. 0x080483db: cmp [ebp+8], 9` instruction to yield the state shown in Figure 4.18.

$$\begin{aligned}\psi_{p_{bb1}} = \Big\{ &\{[ebp+0x8] \geq 0, [esp+0x1c] = 0\},\\ &\{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,\\ &ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x080483df\},\\ &\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},\\ &\{of = 0, sf = 1, zf = 0, af = 1, pf = 1, cf = 1\},\\ &\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,\\ &temp4 = eip + 0x080483e9\}\Big\}\end{aligned}$$

Figure 4.18: State at Basic Block 1 Entrance

With our assumption that our input is between 0 and 9 arguments, we will see that execution of this path will continue to Basic Block 2 (Figure 4.19), then Basic Block 4 (Figure 4.20).

```
.text:080483E1                 mov     dword ptr [esp+1Ch], 0
.text:080483E9                 jmp     short loc_8048415
```

Figure 4.19: *loop-branch* Basic Block 2 (Taken from IDA Pro)

```
.text:08048415 loc_8048415:                            ; CODE XREF: main+25j
.text:08048415                 mov     eax, [esp+1Ch]
.text:08048419                 cmp     eax, [ebp+arg_0]
.text:0804841C                 jl      short loc_80483EB
```

Figure 4.20: *loop-branch* Basic Block 4 (Taken from IDA Pro)

To reach Basic Blocks 2 and 4, their respective states would be as shown in Figure 4.21, and Figure 4.22.

$$\begin{aligned}\psi^0_{p_{bb2}} = \Bigg\{ &\{(([ebp+0x8] \geq 0) \wedge ([ebp+0x8] \leq 9), [esp+0x1c] = 0\},\\ &\{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,\\ &ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x080483e1\},\\ &\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},\\ &\{of = 0, sf = 1, zf = 0, af = 1, pf = 1, cf = 1\},\\ &\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,\\ &temp4 = eip + 0x08048420, temp5 = eip + 0x08048420\}\Bigg\}\end{aligned}$$

Figure 4.21: State at Basic Block 2 Entrance

$$\begin{aligned}\psi^0_{p_{bb4}} = \Bigg\{ &\{(([ebp+0x8] \geq 0) \wedge ([ebp+0x8] \leq 9), [esp+0x1c] = 0\},\\ &\{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,\\ &ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x08048415\},\\ &\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},\\ &\{of = 0, sf = 1, zf = 0, af = 1, pf = 1, cf = 1\},\\ &\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,\\ &temp4 = eip + 0x08048420, temp5 = eip + 0x08048420, temp6 = eip + 0x08048415\}\Bigg\}\end{aligned}$$

Figure 4.22: State at Basic Block 4 Entrance

Since our assumption up to this point has been that we have an input between 0 and 9 arguments, we will have reached Basic Block 7 (Figure 4.23). State ψ_{bb6}, and therefore ϕ_{p_0} at the entrance of Basic Block 7 will be as shown in Figure 4.24.

```
.text:080483EB
.text:080483EB loc_80483EB:                            ; CODE XREF: main+58j
.text:080483EB                 mov     eax, [esp+1Ch]
.text:080483EF                 shl     eax, 2
.text:080483F2                 add     eax, [ebp+arg_4]
.text:080483F5                 mov     edx, [eax]
.text:080483F7                 mov     eax, offset format ; "Hello! argv[%d] = %s\n"
.text:080483FC                 mov     [esp+8], edx
.text:08048400                 mov     edx, [esp+1Ch]
.text:08048404                 mov     [esp+4], edx
.text:08048408                 mov     [esp], eax      ; format
.text:0804840B                 call    _printf
.text:08048410                 add     dword ptr [esp+1Ch], 1
.text:08048415
```

Figure 4.23: *loop-branch* Basic Block 7 (Taken from IDA Pro)

$$\psi^1_{p_{bb7}} = \Big\{ \{(([ebp+0x8] \geq 0) \wedge ([ebp+0x8] \leq 9), [esp+0x1c] = 0\},$$
$$\{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,$$
$$ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x080483eb\},$$
$$\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},$$
$$\{of = 0, sf = 1, zf = 0, af = 1, pf = 1, cf = 1\},$$
$$\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,$$
$$temp4 = eip + 0x08048420, temp5 = eip + 0x08048420,$$
$$temp6 = eip + 0x080483eb\} \Big\}$$

Figure 4.24: State at Basic Block 7 Entrance

From here, we will see that it enters a loop. Detecting loop invariants is beyond the scope of this thesis but at the end of the looping, we can observe the state at the entrance of Basic Block 6 (Figure 4.25) will be as shown in Figure 4.26. We observe that this is the basic block that will output “Hello!” and the argument vector line-by-line per the conditions set when user input was provided.

```
.text:0804841E                 jmp     short loc_804845D
```

Figure 4.25: *loop-branch* Basic Block 6 (Taken from IDA Pro)

$$\psi_{bb6} = \phi_{p_0}^{hello} = \Big\{ \{([ebp+0x8] \geq 0) \wedge ([ebp+0x8] \leq 9), [esp+0x1c] = [ebp+0x8],$$
$$\{eax \in 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0,$$
$$ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x080481eb\},$$
$$\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},$$
$$\{of = 0, sf = 0, zf = 1, af = 0, pf = 1, cf = 0\},$$
$$\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,$$
$$temp4 = eip + 0x08048420, temp5 = eip + 0x08048420,$$
$$temp6 = eip + 0x080483eb\} \Big\}$$

Figure 4.26: State at Basic Block 6 Entrance

From here, an uncondtional jump to Basic Block 9 (Figure 4.27) will take place which will lead the program to exit normally.

```
.text:0804845D loc_804845D:                            ; CODE XREF: main+5Aj
.text:0804845D                 mov     eax, 0
.text:08048462                 leave
.text:08048463                 retn
.text:08048463 main            endp
```

Figure 4.27: *loop-branch* Basic Block 9 (Taken from IDA Pro)

We have just shown an analysis of execution path p where the user input is between 0 and 9 arguments. A similar analysis would follow where the user supplies any positive input count greater than 9. The difference in state output when it enters Basic Block 8 (Figure 4.28) is shown in Figure 4.29

```
.text:0804842A
.text:0804842A loc_804842A:                                ; CODE XREF: main+97j
.text:0804842A                 mov     eax, [esp+1Ch]
.text:0804842E                 shl     eax, 2
.text:08048431                 add     eax, [ebp+arg_4]
.text:08048434                 mov     edx, [eax]
.text:08048436                 mov     eax, offset aGoodbyeArgvDS ; "Goodbye! argv[%d] = %s\n"
.text:0804843B                 mov     [esp+8], edx
.text:0804843F                 mov     edx, [esp+1Ch]
.text:08048443                 mov     [esp+4], edx
.text:08048447                 mov     [esp], eax      ; format
.text:0804844A                 call    _printf
.text:0804844F                 add     dword ptr [esp+1Ch], 1
.text:08048454
```

Figure 4.28: *loop-branch* Basic Block 8 (Taken from IDA Pro)

$$\psi_{bb8} = \phi_{p_0}^{goodbye} = \Big\{ \{(([ebp+0x8] < 0) \wedge ([ebp+0x8] > 9), [esp+0x1c] = 0\},$$
$$\{eax \in 0, ebx = 0, ecx = 0, edx \notin 0:9, edi = 0, esi = 0,$$
$$ebp = 0xbffff6b8, esp = 0xbffff690, eip = 0x0804842a\},$$
$$\{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\},$$
$$\{of = 0, sf = 1, zf = 0, af = 1, pf = 1, cf = 1\},$$
$$\{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0,$$
$$temp4 = eip + 0x08048420, temp5 = eip + 0x08048454,$$
$$temp6 = eip + 0x0804842a\} \Big\}$$

Figure 4.29: State at Basic Block 8 Entrance At Beginning of Loop

4.3.2 Applying Algebraic Semantics to Theorem Proving

We now take the manual analysis above and discuss what should be done to transform it and input it into a theorem prover, such as an SMT or SAT Solver [56]. It is difficult to explicitly describe all the differences between SMT and SAT solvers as the driving theories are intertwined. A rough description for the differences between the two would be analogous to describing the differences between a higher-level programming language such as C and lower-level programming language such as x86 Assembly. SMT can be considered to be built on top of SAT, where SMT utilizes first-order logic [64, 65] to provide a richer language to describe CNF formulas [66–68]. Early attempts to utilize SMT instances involved *eager* encoding the SMT statements to Boolean SAT statements, otherwise known as *bit-blasting* [69]. In many ways, this is analogous to converting C program source code into an assembly listing.

There are advantages and disadvantages to this bit-blasting SMT to SAT. When bit-blasted into Boolean SAT instances, the formulas can be solved by existing SAT solvers without modification. However, since we are going from a higher level language—in this case, first-order logic—to a lower level language, the higher-level semantics of the first-order logic are lost. In this case, the SAT Solver has more reasoning to perform since some of the high-level semantics that seem axiomatically obvious now have to be explicitly stated. One example would be commutative and transitive properties in arithmetic. If we had a formula such as Figure 4.30 and eagerly big-blasted the formula into Boolean SAT, the SAT solver would have to "work harder" to verify the commutative and transitive properties of the formula.

$$x = y \ \&\& \ y = x \ \&\& \ y = z \ \&\& \ z = x$$

Figure 4.30: First-Order Logic Example 1

On the other hand, with SMT, we can use axiomatic statements to enforce the commutative and transitive properties without having the SMT solver work to try to verify them since we have postulated that the properties simply hold. Figure 4.31 shows an example of this.

$$\text{\textbackslash forall}\, x, y, z.\ x = y \ \&\& \ y = x \ \&\& \ y = z \ \&\& \ z = x$$

Figure 4.31: First-Order Logic Example 2—Axiomatic Enforcement

To begin, we must observe that our framework currently utilizes just-in-time symbolic execution as described in Chapter 3. That is, the symbolic execution engine does not have apriori knowledge of the binary code under test other than what it knows from loading the program into memory. From here, we must encode the initial state of the program.

It may be necessary to encode the state of every instruction to retain stateful history of all instructions executed for granular debugging of a program. However, it may only be necessary to generate SMT formulas that test and asserts a subset of constraints that will help us determine a property of execution flow. It is our contention that we want the theorem prover to look at instructions that perform “conditional” tests. For instance, if our emulation engine retrieves a `test` or `cmp` instruction, we want to determine the state from that instruction and then use the state to encode what needs to be tested.

While other analyses are important, such as properly identifying basic blocks, identifying loops, and determining value ranges, these are “big-picture” issues that the emulation engine will handle with control flow and data flow analysis. As discussed in Section 4.3, we observed that we were able to go down two major paths of execution as shown in Figure 4.8. This indicates that there were at least two paths of execution to traverse. This means that at least two execution traces can be made for *loop-branch*. Each trace, up to some point of execution, will be encoded into SMT/SAT compatible formulas [66–68] to determine path reachability.

For the purposes of the discussion in this section, we will focus on *SMT* encoding. Specifically, we will be focusing on the SMT-LIB 2.0 standard [66].

Handling the analysis of binary code under test means that we will not be dealing with source code as discussed. To that end, we will rely on logic dealing with *bit-vectors* and arrays. A bit-vector is an array data structure of Boolean values of a given length [57]. This is relevant since we are dealing with programs at the bit level and our reasoning of native x86 instructions requires the ability to perform bit-wise operations. In general, we will use *QF_AUFBV* logic in SMT. When we refer to a *logic* in SMT, we are referring to the theories, functions, and symbols that are associated with a logic. A *theory*, in the context of SMT solvers, defines a vocabulary of sorts (otherwise known as data types) and functions, and it associates a sort (type) with relevant literals. *QF_AUFBV*, in this context, refers to **Quantifier-Free with Arrays, Uninterpreted Functions, and Bit-Vectors** [67, 68], which allows quantifier-free expressions, including the family of bit-vector sorts and functions associated with fixed size bit-vector theory, arrays, arbitrary sorts with uninterpreted functions, and associated function symbols [68].

Revisiting our *loop-branch* example, let us begin a discussion of what are relevant points to call the theorem prover. In Section 4.3, a special state was introduced—$S_{!BRANCH!}$—to indicate a jump target or unconditional jump. This state was explicitly defined in order to indicate when a branch would occur. We will focus on Basic Block 0 (Figure 4.32).

```
.text:080483C4 ; =============== S U B R O U T I N E =======================================
.text:080483C4
.text:080483C4 ; Attributes: bp-based frame
.text:080483C4
.text:080483C4                 public main
.text:080483C4 main            proc near               ; DATA XREF: _start+17o
.text:080483C4
.text:080483C4 arg_0           = dword ptr  8
.text:080483C4 arg_4           = dword ptr  0Ch
.text:080483C4
.text:080483C4                 push    ebp
.text:080483C5                 mov     ebp, esp
.text:080483C7                 and     esp, 0FFFFFFF0h
.text:080483CA                 sub     esp, 20h
.text:080483CD                 mov     dword ptr [esp+1Ch], 0
.text:080483D5                 cmp     [ebp+arg_0], 0
.text:080483D9                 js      short loc_8048420
```

Figure 4.32: *loop-branch* Basic Block 0 (Taken from IDA Pro)

We have mathematically represented the instruction encoding for each instruction that allowed the program counter to reach the `0x080483d9: js short loc_8048420` instruction. We have further represented the states that are reachable when the branch is taken and when the branch is not taken, as shown in Figure 4.33 and Figure 4.34.

$$
\begin{aligned}
\psi^0_{bb0} = \phi^0_{p_0} = \Big\{ & \{[ebp+0x8] \geq 0, [esp+0x1c] = 0\}, \\
& \{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0, \\
& ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x080483db\}, \\
& \{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\}, \\
& \{of = 0, sf = 0, zf = 0, af = 0, pf = 0, cf = 0\}, \\
& \{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0, \\
& temp4 = eip + 0x080483db\} \Big\}
\end{aligned}
$$

Figure 4.33: State that causes Basic Block 0 to Branch to Basic Block 1

$$
\begin{aligned}
\psi^1_{bb0} = \phi^1_{p_0} = \Bigg\{ & \{[ebp+0x8] < 0, [esp+0x1c] = 0\}, \\
& \{eax = 0, ebx = 0, ecx = 0, edx = 0, edi = 0, esi = 0, \\
& ebp = 0xbffff6c8, esp = 0xbffff6a0, eip = 0x08048420\}, \\
& \{cs = 0x73, ds = 0x7b, ss = 0x7b, es = 0x7b, fs = 0x0, gs = 0x33\}, \\
& \{of = 0, sf = 1, zf = 0, af = 0, pf = 0, cf = 0\}, \\
& \{temp1 = ebp, temp2 = 0xfffffff0, temp3 = [ebp+8] - 0, \\
& temp4 = eip + 0x08048420\} \Bigg\}
\end{aligned}
$$

Figure 4.34: State that causes Basic Block 0 to Branch to Basic Block 3

We can represent the registers as 32-bit data structures. In SMT-LIB, they would be represented with the bit-vector sort `(_ BitVec 32)`. The generic statement for a bit-vector of arbitrary length `n` is `(_ BitVec n)`, which allows us to specify bit-vectors of other sizes. Bit-vector literals can also be defined using binary, decimal, and hexadecimal notation. Since we are using hexadecimal representations for memory addressing, it makes sense to stay consistent. An example of using hexadecimal notation would be `#xbffff6c8` to represent the hexadecimal address `0xbffff6c8`.

Using our knowledge of SMT-LIB, we can proceed to encode the state in a meaningful manner. The encodings are showing in Figure 4.35 and Figure 4.36.

For the purposes of illustrating how a theorem prover is utilized, assume that the total number of user-supplied arguments was five. We would then be able to encode an SMT instance as shown in Figure 4.35.

```
; set logic to quantifier free with arrays,
; uninterpreted functions, and bit-vectors
(set-logic QF_AUFBV)

; declare variables
(define-fun eax () (_ BitVec 32) #x00000000)
(define-fun ebx () (_ BitVec 32) #x00000000)
(define-fun ecx () (_ BitVec 32) #x00000000)
(define-fun edx () (_ BitVec 32) #x00000000)
(define-fun edi () (_ BitVec 32) #x00000000)
(define-fun esi () (_ BitVec 32) #x00000000)
(define-fun ebp () (_ BitVec 32) #xbffff6c8)
(define-fun esp () (_ BitVec 32) #xbffff6a0)
(define-fun eip () (_ BitVec 32) #x080483db)
(define-fun cs () (_ BitVec 8) #x73)
(define-fun ds () (_ BitVec 8) #x7b)
(define-fun ss () (_ BitVec 8) #x7b)
(define-fun es () (_ BitVec 8) #x7b)
(define-fun fs () (_ BitVec 8) #x99)
(define-fun gs () (_ BitVec 8) #x33)
(define-fun of () (_ BitVec 4) #x0)
(define-fun sf () (_ BitVec 4) #x0)
(define-fun zf () (_ BitVec 4) #x0)
(define-fun af () (_ BitVec 4) #x0)
(define-fun pf () (_ BitVec 4) #x0)
(define-fun cf () (_ BitVec 4) #x0)
(define-fun temp1 () (_ BitVec 32) #x00000005) ;temp value for [ebp+0x8]=5
(define-fun temp2 () (_ BitVec 32) #x00000000) ;temp value for comparison value against [ebp+0x8]
(define-fun temp3 () (_ BitVec 32) #x00000000) ;temp value for [esp+0x1c]
(define-fun temp4 () (_ BitVec 32) #x00000000) ;temp value forcomparison value against [esp+0x1c]

; test: (signed) temp1 >= temp2 && temp3 == temp4
(assert (and (bvsge temp1 temp2) (= temp3 temp4)))

; instantiate smt solver
(check-sat)
```

Figure 4.35: SMT encoding of ψ^{0}_{bb0}

If we managed to somehow pass a negative number of total arguments, then we would have an SMT instance as shown in Figure 4.36. Assume -5 arguments were sent.

```
; set logic to quantifier free with arrays,
; uninterpreted functions, and bit-vectors
(set-logic QF_AUFBV)

; declare variables
(define-fun eax () (_ BitVec 32) #x00000000)
(define-fun ebx () (_ BitVec 32) #x00000000)
(define-fun ecx () (_ BitVec 32) #x00000000)
(define-fun edx () (_ BitVec 32) #x00000000)
(define-fun edi () (_ BitVec 32) #x00000000)
(define-fun esi () (_ BitVec 32) #x00000000)
(define-fun ebp () (_ BitVec 32) #xbffff6c8)
(define-fun esp () (_ BitVec 32) #xbffff6a0)
(define-fun eip () (_ BitVec 32) #x08048420)
(define-fun cs () (_ BitVec 8) #x73)
(define-fun ds () (_ BitVec 8) #x7b)
(define-fun ss () (_ BitVec 8) #x7b)
(define-fun es () (_ BitVec 8) #x7b)
(define-fun fs () (_ BitVec 8) #x99)
(define-fun gs () (_ BitVec 8) #x33)
(define-fun of () (_ BitVec 4) #x0)
(define-fun sf () (_ BitVec 4) #x1)
(define-fun zf () (_ BitVec 4) #x0)
(define-fun af () (_ BitVec 4) #x0)
(define-fun pf () (_ BitVec 4) #x0)
(define-fun cf () (_ BitVec 4) #x0)
(define-fun temp1 () (_ BitVec 32) #xfffffffb) ;temp value for [ebp+0x8]=-5
(define-fun temp2 () (_ BitVec 32) #x00000000) ;temp value for comparison value against [ebp+0x8]
(define-fun temp3 () (_ BitVec 32) #x00000000) ;temp value for [esp+0x1c]
(define-fun temp4 () (_ BitVec 32) #x00000000) ;temp value forcomparison value against [esp+0x1c]

; test: (signed) temp1 < temp2 && temp3 == temp4
(assert (and (bvslt temp1 temp2) (= temp3 temp4)))

; instantiate smt solver
(check-sat)
```

Figure 4.36: SMT encoding of ψ^1_{bb0}

With these SMT instances, we could then verify our conditions held and we could then ask the question of whether a given execution path is reachable or not. Despite the results produced by a theorem prover, human interaction and logic must be used here. As we can see from the above example, it is certainly logical that execution could branch from Basic Block 0 to Basic Block 3 if we pass a negative number of arguments. However, this may not be a feasible path to visit since it is not possible to pass a negative number of arguments.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions and Future Work

5.1 Conclusions and Discussions

This thesis asked the following question: Can an algebra be developed to mathematically describe the state of binary code during execution? Further, can this algebra describe program state at the instruction level, basic block level, and the program level. To answer this question, an algebra was indeed developed that would precisely describe program state as a program executed.

We began by using Tseitin transformation principles to perform encoding at the instruction level and describe the operation an Intel x86 instruction performs on its respective operands. This was done using set theory to help mathematically describe the encoding to conjunctive normal form. Additionally, we observed bit-wise operations that occur for the instructions.

We then expanded our state definition to the basic block level with the goal of describing the state of execution at both the entrance and exit of a basic block. With our developed algebra, one implicit byproduct was the capability of describing the state at any instruction within a basic block. However, the main goal was to effectively mathematically model entrance and exit state context to satisfy the objective of path reconstruction (will be discussed in future work).

Finally, we created a definition that modeled the overall program state based on our earlier knowledge garnered from encoding each encountered instruction and each traversed basic block respective to a particular execution path. The ability to distinguish unique execution paths is critical to successful program analysis. Even simple programs can produce several paths of execution, and subsequently many iterations of state, all of which must be explored for complete and exhaustive code coverage. Unfortunately, this leads to the problem of state explosion (will be discussed in future work).

With respect to developing an algebra that mathematically modeled the state of binary code under execution, we were successful. The modeling process is meant to be a method that can be directly implemented under our analysis framework. That objective greatly influenced how the algebra was initially developed. Further, we believe that there are still methods to improve and optimize the modeling process.

We must observe that despite the seemingly illogical results that the theorem prover can give us, it is because of the question we are asking. We provide an instance that effectively is asking, “Given the constraints we have found, could we reach this path?” Indeed, it would perhaps be more useful if we structured our analysis to come upon conditionals and other points of interest within program execution and use the knowledge garnered from the analysis to ask, “Given the constraints we have found, where can we go?”

5.2 Future Work

This thesis provides a baseline for mathematically modeling program state for code under execution, a brief overview of related efforts for framework development, and discussion on how the algebraic notation can be used to encode Boolean formulas for proving in a theorem prover such as an SMT or SAT solver. However, this is just scratching the surface in the field of program analysis and there are many efforts that can expand this work in many ways.

CONTROL FLOW AND DATA FLOW ANALYSIS. The heart of our framework is its ability to perform robust analysis. Further research into control flow analysis is necessary to increase our framework’s capability to analyze decision points in a program effectively. One area, in particular is the ability to detect loop invariants [70]. Detecting loops is a hard problem to solve and a good deal of research has come out discussing methods to infer loops during analysis of code under execution. The ability to infer the possibility of loops would be useful for implementation into our framework. A starting point for other control flow analysis elements to research for implementation can be found in Muchnick [21]

Data flow analysis is a method of gathering information on how functions manipulate data and keep track of values [21]. Data flow analysis can utilize the same CFG that would be used in control flow analysis. Data flow analysis is critically important as taint analysis is a type of data flow analysis since taint analysis is a method of keeping track of user-supplied data that propagates through a process. Elements of data flow analysis, including reaching definitions, interprocedural analysis, and shape analysis, both discussed in Nielson [11], are interesting areas to look into further to increase the robustness of our analysis engine.

STATIC SINGLE ASSIGNMENT (SSA). Static Single Assignment (SSA) [63] is an effective concept that is currently applied to compiler theory and program analysis. As of now, our algebra and engine only use SSA for creating unique variables as they are encountered during code execution and analysis. It may be useful to research how to further SSA concepts to optimize the modeling and analysis process. One key concept in SSA is the use of phi functions. How to effectively use these in our framework may reduce the overall amount of computation necessary.

ALGORITHMS FOR BACK TRACING. Reinbacher and Brauer [53] provide very interesting methods of reconstructing execution flows using forward and backward interpretation (tracing) to. One of our primary goals is to be able to precisely trace back paths of execution that are interesting to us for analysis—specifically paths that lead to program bugs. Research into using our algebraically derived state to perform backtracking can be useful for identifying executions paths of interest. For example, if the emulation/analysis engine executes down a path that results in identification of a vulnerability, then being able to perform path reconstruction would of value.

OVER- AND UNDER-APPROXIMATION. Cousot introduced *Abstract Interpretation* to perform program analysis by approximating the semantic behavior of a program [38, 71]. Symbolic execution can be considered a sub-case of abstract interpretation. One particular idea to come from this field of program analysis is the idea of over- and under-approximation in order to manage the amount of state exploration needed to find execution paths of interest. Research into this would be interesting to help optimize the analysis framework and increase productivity in finding execution paths that are of interest. Additionally, one must consider the advantages and disadvantages of both over-approximation and under-approximation with regards to precision, accuracy, exhaustion of state exploration, time of analysis, and resource allocation.

COUPLING FUZZING/CONCOLIC TESTING. Concolic testing [5, 6, 24, 72–74] is a hybrid method of testing that mixes both symbolic analysis and concrete analysis. The goal of coupling both is the ability to conduct testing [3] in lockstep with the symbolic execution engine to narrow down possible code coverage and minimize state explosion. Work should be done to combine the capabilities of both symbolic analysis and some variant of concrete or semi-concrete analysis, such as grammar-based fuzzing, to improve the ability to test programs for interesting execution paths.

IMPLEMENTATION OF FORMAL ALGEBRA FOR THEOREM PROVING. This thesis has presented a formal algebra for representing the state of binary code under test and analysis. It has also presented a short discussion of the necessary steps to encode the mathematical representation of this contextual state at the instruction, basic block, and program level to SMT-compatible first-order logic formulae in order to feed it to an SMT solver or for bit-blasting the SMT formulas into Boolean SAT instances for inputting into a SAT solver in Chapter 4. Further work must be done to fully implement this into our framework.

Ultimately our primary question, given our goals, is to ask the question, “Is this path reachable, given our analysis?” This is the fundamental question we ask when we consider the execution and analysis that the emulation engine performs and feeds to the algebra. The example of encoding SMT instances in Chapter 4 was but a simple and verbose example of testing a condition that would result in a branch. Optimizing the method of generating SMT or SAT instances would be advantageous simply to minimize resources for processing theorems as they are fed. As alluded to above, however, we may want to structure our analysis to instead ask, “Given the constraints we have found, where can we go?” Additional efforts should be made to see if we need to encode state in the manner we are currently doing so. For example, if there are registers and memory addresses that are not part of a Boolean instance to determine reachability, perhaps it is not necessary to encode those in a given Boolean instance.

Additionally, research must be done to optimize the points of execution where a theorem prover must be called. Specifically, research should be done to ascertain whether it is necessary to always encode every instruction, or whether this process could be reduced by using state derived from each basic block to determine path reachability.

MODULAR THEOREM PROVING. SMT/SAT Solving, as discussed in Chapter 2 and Chapter 3, is a large field of research because of the many intricacies associated with satisfiability as applied to path reachability and code coverage. The mathematical algebra developed in this thesis relates to the theorem proving capability as the algebra must translate the program context into Boolean arguments that will be fed into a SMT/SAT solver. Our framework would benefit from research on creating and optimizing the ability to use different solvers in an automated fashion.

It is hoped that this thesis will contribute ideas that will benefit the research community in further progressing the state of the art in program analysis and bug hunting.

APPENDIX A:
Program Listing: tree.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void print_out(const char *string) {
    printf("%s", string);
}

int main(int argc, char *argv[]) {
    char *true_statement = "This statement is true!\n";
    char *false_statement = "This statement is false!\n";

    int i = 0;

    if (argc == 1) {
        printf("boo moar args please nom nom!\n");
    } else if (argc == 2) {
        for(i = 0; i < argc; i++) {
            if (i % 2 == 1) {
                print_out(false_statement);
            } else {
                print_out(true_statement);
            }
        }
    } else if (argc == 3) {
        for(i = 0; i < argc; i++) {
            if (i % 2 == 1) {
                print_out(false_statement);
            } else {
                print_out(true_statement);
            }
        }
    } else if (argc > 3) {
        for(i = 0; i < argc; i++) {
            if (i % 2 == 1) {
                print_out(false_statement);
            } else {
                print_out(true_statement);
            }
        }
    }
    return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Disassembly Listing: tree

```
.init:080482B4 ;
.init:080482B4 ; +-------------------------------------------------------------------------+
.init:080482B4 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
.init:080482B4 ; |           Copyright (c) 2011 Hex-Rays, <support@hex-rays.com>           |
.init:080482B4 ; |                      License info: XXXXXXXXXXXXXXX                      |
.init:080482B4 ; |                            XXXXXXXXXXXXXXXXX                            |
.init:080482B4 ; +-------------------------------------------------------------------------+
.init:080482B4 ;
.init:080482B4 ; Input MD5   : 4EFF7FC09CD531AA2A0983B4B09BB8BE
.init:080482B4 ; Input CRC32 : CC031A6A
.init:080482B4
.init:080482B4 ; File Name   : tree
.init:080482B4 ; Format      : ELF for Intel 386 (Executable)
.init:080482B4 ; Imagebase   : 8048000
.init:080482B4 ; Interpreter '/lib/ld-linux.so.2'
.init:080482B4 ; Needed Library 'libc.so.6'
.init:080482B4 ;
.init:080482B4 ; Source File : 'crtstuff.c'
.init:080482B4 ; Source File : 'tree.c'
.init:080482B4
.init:080482B4                 .686p
.init:080482B4                 .mmx
.init:080482B4                 .model flat
.init:080482B4 .intel_syntax noprefix

.. CODE SNIPPET ..

.text:080483F4 ; =============== S U B R O U T I N E =======================================
.text:080483F4
.text:080483F4 ; Attributes: bp-based frame
.text:080483F4
.text:080483F4                 public print_out
.text:080483F4 print_out       proc near               ; CODE XREF: main+67p
.text:080483F4                                         ; main+75Ep ...
.text:080483F4
.text:080483F4 arg_0           = dword ptr  8
.text:080483F4
.text:080483F4                 push    ebp
.text:080483F5                 mov     ebp, esp
.text:080483F7                 sub     esp, 18h
.text:080483FA                 mov     eax, offset format ; "%s"
.text:080483FF                 mov     edx, [ebp+arg_0]
.text:08048402                 mov     [esp+4], edx
.text:08048406                 mov     [esp], eax      ; format
.text:08048409                 call    _printf
.text:0804840E                 leave
```

```
.text:0804840F                 retn
.text:0804840F print_out       endp
.text:0804840F
.text:08048410
.text:08048410 ; =============== S U B R O U T I N E =======================================
.text:08048410
.text:08048410 ; Attributes: bp-based frame
.text:08048410
.text:08048410                 public main
.text:08048410 main            proc near               ; DATA XREF: _start+17o
.text:08048410
.text:08048410 arg_0           = dword ptr  8
.text:08048410
.text:08048410                 push    ebp
.text:08048411                 mov     ebp, esp
.text:08048413                 and     esp, 0FFFFFFF0h
.text:08048416                 sub     esp, 20h
.text:08048419                 mov     dword ptr [esp+14h], offset aThisStatementI ; "This statement is true!\n"
.text:08048421                 mov     dword ptr [esp+18h], offset aThisStatemen_0 ; "This statement is false!\n"
.text:08048429                 mov     dword ptr [esp+1Ch], 0
.text:08048431                 cmp     [ebp+arg_0], 1
.text:08048435                 jnz     short loc_8048448
.text:08048437                 mov     dword ptr [esp], offset s ; "boo moar args please nom nom!"
.text:0804843E                 call    _puts
.text:08048443                 jmp     loc_804853F
.text:08048448 ; ---------------------------------------------------------------------------
.text:08048448
.text:08048448 loc_8048448:                            ; CODE XREF: main+25j
.text:08048448                 cmp     [ebp+arg_0], 2
.text:0804844C                 jnz     short loc_804849D
.text:0804844E                 mov     dword ptr [esp+1Ch], 0
.text:08048456                 jmp     short loc_804848F
.text:08048458 ; ---------------------------------------------------------------------------
.text:08048458
.text:08048458 loc_8048458:                            ; CODE XREF: main+86j
.text:08048458                 mov     eax, [esp+1Ch]
.text:0804845C                 mov     edx, eax
.text:0804845E                 sar     edx, 1Fh
.text:08048461                 shr     edx, 1Fh
.text:08048464                 add     eax, edx
.text:08048466                 and     eax, 1
.text:08048469                 sub     eax, edx
.text:0804846B                 cmp     eax, 1
.text:0804846E                 jnz     short loc_804847E
.text:08048470                 mov     eax, [esp+18h]
.text:08048474                 mov     [esp], eax
.text:08048477                 call    print_out
.text:0804847C                 jmp     short loc_804848A
.text:0804847E ; ---------------------------------------------------------------------------
.text:0804847E
.text:0804847E loc_804847E:                            ; CODE XREF: main+5Ej
.text:0804847E                 mov     eax, [esp+14h]
```

```
.text:08048482                 mov     [esp], eax
.text:08048485                 call    print_out
.text:0804848A
.text:0804848A loc_804848A:                            ; CODE XREF: main+6Cj
.text:0804848A                 add     dword ptr [esp+1Ch], 1
.text:0804848F
.text:0804848F loc_804848F:                            ; CODE XREF: main+46j
.text:0804848F                 mov     eax, [esp+1Ch]
.text:08048493                 cmp     eax, [ebp+arg_0]
.text:08048496                 jl      short loc_8048458
.text:08048498                 jmp     loc_804853F
.text:0804849D ; ---------------------------------------------------------------------------
.text:0804849D
.text:0804849D loc_804849D:                            ; CODE XREF: main+3Cj
.text:0804849D                 cmp     [ebp+arg_0], 3
.text:080484A1                 jnz     short loc_80484EF
.text:080484A3                 mov     dword ptr [esp+1Ch], 0
.text:080484AB                 jmp     short loc_80484E4
.text:080484AD ; ---------------------------------------------------------------------------
.text:080484AD
.text:080484AD loc_80484AD:                            ; CODE XREF: main+DBj
.text:080484AD                 mov     eax, [esp+1Ch]
.text:080484B1                 mov     edx, eax
.text:080484B3                 sar     edx, 1Fh
.text:080484B6                 shr     edx, 1Fh
.text:080484B9                 add     eax, edx
.text:080484BB                 and     eax, 1
.text:080484BE                 sub     eax, edx
.text:080484C0                 cmp     eax, 1
.text:080484C3                 jnz     short loc_80484D3
.text:080484C5                 mov     eax, [esp+18h]
.text:080484C9                 mov     [esp], eax
.text:080484CC                 call    print_out
.text:080484D1                 jmp     short loc_80484DF
.text:080484D3 ; ---------------------------------------------------------------------------
.text:080484D3
.text:080484D3 loc_80484D3:                            ; CODE XREF: main+B3j
.text:080484D3                 mov     eax, [esp+14h]
.text:080484D7                 mov     [esp], eax
.text:080484DA                 call    print_out
.text:080484DF
.text:080484DF loc_80484DF:                            ; CODE XREF: main+C1j
.text:080484DF                 add     dword ptr [esp+1Ch], 1
.text:080484E4
.text:080484E4 loc_80484E4:                            ; CODE XREF: main+9Bj
.text:080484E4                 mov     eax, [esp+1Ch]
.text:080484E8                 cmp     eax, [ebp+arg_0]
.text:080484EB                 jl      short loc_80484AD
.text:080484ED                 jmp     short loc_804853F
.text:080484EF ; ---------------------------------------------------------------------------
.text:080484EF
.text:080484EF loc_80484EF:                            ; CODE XREF: main+91j
```

```
.text:080484EF                 cmp     [ebp+arg_0], 3
.text:080484F3                 jle     short loc_804853F
.text:080484F5                 mov     dword ptr [esp+1Ch], 0
.text:080484FD                 jmp     short loc_8048536
.text:080484FF ; ---------------------------------------------------------------------------
.text:080484FF
.text:080484FF loc_80484FF:                            ; CODE XREF: main+12Dj
.text:080484FF                 mov     eax, [esp+1Ch]
.text:08048503                 mov     edx, eax
.text:08048505                 sar     edx, 1Fh
.text:08048508                 shr     edx, 1Fh
.text:0804850B                 add     eax, edx
.text:0804850D                 and     eax, 1
.text:08048510                 sub     eax, edx
.text:08048512                 cmp     eax, 1
.text:08048515                 jnz     short loc_8048525
.text:08048517                 mov     eax, [esp+18h]
.text:0804851B                 mov     [esp], eax
.text:0804851E                 call    print_out
.text:08048523                 jmp     short loc_8048531
.text:08048525 ; ---------------------------------------------------------------------------
.text:08048525
.text:08048525 loc_8048525:                            ; CODE XREF: main+105j
.text:08048525                 mov     eax, [esp+14h]
.text:08048529                 mov     [esp], eax
.text:0804852C                 call    print_out
.text:08048531
.text:08048531 loc_8048531:                            ; CODE XREF: main+113j
.text:08048531                 add     dword ptr [esp+1Ch], 1
.text:08048536
.text:08048536 loc_8048536:                            ; CODE XREF: main+EDj
.text:08048536                 mov     eax, [esp+1Ch]
.text:0804853A                 cmp     eax, [ebp+arg_0]
.text:0804853D                 jl      short loc_80484FF
.text:0804853F
.text:0804853F loc_804853F:                            ; CODE XREF: main+33j
.text:0804853F                                         ; main+88j ...
.text:0804853F                 mov     eax, 0
.text:08048544                 leave
.text:08048545                 retn
.text:08048545 main            endp
.text:08048545
.text:08048545 ; ---------------------------------------------------------------------------
.text:08048546                 align 10h

.. CODE SNIPPET ..

.rodata:08048608 ; ===========================================================================
.rodata:08048608
.rodata:08048608 ; Segment type: Pure data
.rodata:08048608 ; Segment permissions: Read
.rodata:08048608 _rodata         segment dword public 'CONST' use32
```

```
.rodata:08048608                 assume cs:_rodata
.rodata:08048608                 ;org 8048608h
.rodata:08048608                 public _fp_hw
.rodata:08048608 _fp_hw          dd 3
.rodata:0804860C                 public _IO_stdin_used
.rodata:0804860C _IO_stdin_used  dd 20001h
.rodata:08048610                 public __dso_handle
.rodata:08048610 __dso_handle    db    0
.rodata:08048611                 db    0
.rodata:08048612                 db    0
.rodata:08048613                 db    0
.rodata:08048614 ; char format[3]
.rodata:08048614 format          db '%s',0               ; DATA XREF: print_out+6o
.rodata:08048617 aThisStatementI db 'This statement is true!',0Ah,0 ; DATA XREF: main+9o
.rodata:08048630 aThisStatemen_0 db 'This statement is false!',0Ah,0 ; DATA XREF: main+11o
.rodata:0804864A ; char s[]
.rodata:0804864A s               db 'boo moar args please nom nom!',0 ; DATA XREF: main+27o
.rodata:0804864A _rodata         ends
.rodata:0804864A

.. CODE SNIPPET ..
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
Program Listing: simple-branch.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    unsigned int x;

    x = argc;
    if (x >= 0 && x <= 9) {
        printf("Hello!\n");
    } else {
        printf("Goodbye!\n");
    }
    return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
Disassembly Listing: simple-branch

```
.init:08048290 ;
.init:08048290 ; +-------------------------------------------------------------------------+
.init:08048290 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
.init:08048290 ; |           Copyright (c) 2011 Hex-Rays, <support@hex-rays.com>           |
.init:08048290 ; |                      License info: XXXXXXXXXXXXXXX                      |
.init:08048290 ; |                           XXXXXXXXXXXXXXXXX                             |
.init:08048290 ; +-------------------------------------------------------------------------+
.init:08048290 ;
.init:08048290 ; Input MD5   : 38AA512B47816C6DA5AE8356A2099013
.init:08048290 ; Input CRC32 : B1D4A87C
.init:08048290
.init:08048290 ; File Name   : simple-branch
.init:08048290 ; Format      : ELF for Intel 386 (Executable)
.init:08048290 ; Imagebase   : 8048000
.init:08048290 ; Interpreter '/lib/ld-linux.so.2'
.init:08048290 ; Needed Library 'libc.so.6'
.init:08048290 ;
.init:08048290 ; Source File : 'crtstuff.c'
.init:08048290 ; Source File : 'simple-branch.c'
.init:08048290
.init:08048290                 .686p
.init:08048290                 .mmx
.init:08048290                 .model flat
.init:08048290 .intel_syntax noprefix

.. CODE SNIPPET ..

.text:080483B4 ; =============== S U B R O U T I N E =======================================
.text:080483B4
.text:080483B4 ; Attributes: bp-based frame
.text:080483B4
.text:080483B4                 public main
.text:080483B4 main            proc near               ; DATA XREF: _start+17o
.text:080483B4
.text:080483B4 arg_0           = dword ptr  8
.text:080483B4
.text:080483B4                 push    ebp
.text:080483B5                 mov     ebp, esp
.text:080483B7                 and     esp, 0FFFFFFF0h
.text:080483BA                 sub     esp, 20h
.text:080483BD                 mov     eax, [ebp+arg_0]
.text:080483C0                 mov     [esp+1Ch], eax
.text:080483C4                 cmp     dword ptr [esp+1Ch], 9
.text:080483C9                 ja      short loc_80483D9
.text:080483CB                 mov     dword ptr [esp], offset s ; "Hello!"
.text:080483D2                 call    _puts
```

```
.text:080483D7                 jmp     short loc_80483E5
.text:080483D9 ; ---------------------------------------------------------------------------
.text:080483D9
.text:080483D9 loc_80483D9:                            ; CODE XREF: main+15j
.text:080483D9                 mov     dword ptr [esp], offset aGoodbye ; "Goodbye!"
.text:080483E0                 call    _puts
.text:080483E5
.text:080483E5 loc_80483E5:                            ; CODE XREF: main+23j
.text:080483E5                 mov     eax, 0
.text:080483EA                 leave
.text:080483EB                 retn
.text:080483EB main            endp


.. CODE SNIPPET ..

.rodata:080484A8 ; ===========================================================================
.rodata:080484A8
.rodata:080484A8 ; Segment type: Pure data
.rodata:080484A8 ; Segment permissions: Read
.rodata:080484A8 _rodata         segment dword public 'CONST' use32
.rodata:080484A8                 assume cs:_rodata
.rodata:080484A8                 ;org 80484A8h
.rodata:080484A8                 public _fp_hw
.rodata:080484A8 _fp_hw          dd 3
.rodata:080484AC                 public _IO_stdin_used
.rodata:080484AC _IO_stdin_used  dd 20001h
.rodata:080484B0                 public __dso_handle
.rodata:080484B0 __dso_handle    db    0
.rodata:080484B1                 db    0
.rodata:080484B2                 db    0
.rodata:080484B3                 db    0
.rodata:080484B4 ; char s[]
.rodata:080484B4 s               db 'Hello!',0           ; DATA XREF: main+17o
.rodata:080484BB ; char aGoodbye[]
.rodata:080484BB aGoodbye        db 'Goodbye!',0         ; DATA XREF: main:loc_80483D9o
.rodata:080484BB _rodata         ends
.rodata:080484BB

.. CODE SNIPPET ..
```

APPENDIX E:
Program Listing: loop-branch.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int i = 0;

    if(argc >= 0 && argc <= 9) {
        for (i = 0; i < argc; i++) {
            printf("Hello! argv[%d] = %s\n", i, argv[i]);
        }
    } else {
        for (i = 0; i < argc; i++) {
            printf("Goodbye! argv[%d] = %s\n", i, argv[i]);
        }
    }
    return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F: Disassembly Listing: loop-branch

```
.init:08048294 ;
.init:08048294 ; +-------------------------------------------------------------------------+
.init:08048294 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
.init:08048294 ; |           Copyright (c) 2011 Hex-Rays, <support@hex-rays.com>           |
.init:08048294 ; |                       License info: XXXXXXXXXXXXXXX                     |
.init:08048294 ; |                             XXXXXXXXXXXXXXXXX                           |
.init:08048294 ; +-------------------------------------------------------------------------+
.init:08048294 ;
.init:08048294 ; Input MD5   : 6E132184033BD59125FE1D6F54649532
.init:08048294 ; Input CRC32 : FB45EB9B
.init:08048294
.init:08048294 ; File Name   : loop-branch
.init:08048294 ; Format      : ELF for Intel 386 (Executable)
.init:08048294 ; Imagebase   : 8048000
.init:08048294 ; Interpreter '/lib/ld-linux.so.2'
.init:08048294 ; Needed Library 'libc.so.6'
.init:08048294 ;
.init:08048294 ; Source File : 'crtstuff.c'
.init:08048294 ; Source File : 'loop-branch.c'
.init:08048294
.init:08048294                 .686p
.init:08048294                 .mmx
.init:08048294                 .model flat
.init:08048294 .intel_syntax noprefix

.. CODE SNIPPET ..

.text:080483C4 ; =============== S U B R O U T I N E =======================================
.text:080483C4
.text:080483C4 ; Attributes: bp-based frame
.text:080483C4
.text:080483C4                 public main
.text:080483C4 main            proc near               ; DATA XREF: _start+17o
.text:080483C4
.text:080483C4 arg_0           = dword ptr  8
.text:080483C4 arg_4           = dword ptr  0Ch
.text:080483C4
.text:080483C4                 push    ebp
.text:080483C5                 mov     ebp, esp
.text:080483C7                 and     esp, 0FFFFFFF0h
.text:080483CA                 sub     esp, 20h
.text:080483CD                 mov     dword ptr [esp+1Ch], 0
.text:080483D5                 cmp     [ebp+arg_0], 0
.text:080483D9                 js      short loc_8048420
.text:080483DB                 cmp     [ebp+arg_0], 9
.text:080483DF                 jg      short loc_8048420
```

```
.text:080483E1                 mov     dword ptr [esp+1Ch], 0
.text:080483E9                 jmp     short loc_8048415
.text:080483EB ; ---------------------------------------------------------------------------
.text:080483EB
.text:080483EB loc_80483EB:                            ; CODE XREF: main+58j
.text:080483EB                 mov     eax, [esp+1Ch]
.text:080483EF                 shl     eax, 2
.text:080483F2                 add     eax, [ebp+arg_4]
.text:080483F5                 mov     edx, [eax]
.text:080483F7                 mov     eax, offset format ; "Hello! argv[%d] = %s\n"
.text:080483FC                 mov     [esp+8], edx
.text:08048400                 mov     edx, [esp+1Ch]
.text:08048404                 mov     [esp+4], edx
.text:08048408                 mov     [esp], eax      ; format
.text:0804840B                 call    _printf
.text:08048410                 add     dword ptr [esp+1Ch], 1
.text:08048415
.text:08048415 loc_8048415:                            ; CODE XREF: main+25j
.text:08048415                 mov     eax, [esp+1Ch]
.text:08048419                 cmp     eax, [ebp+arg_0]
.text:0804841C                 jl      short loc_80483EB
.text:0804841E                 jmp     short loc_804845D
.text:08048420 ; ---------------------------------------------------------------------------
.text:08048420
.text:08048420 loc_8048420:                            ; CODE XREF: main+15j
.text:08048420                                         ; main+1Bj
.text:08048420                 mov     dword ptr [esp+1Ch], 0
.text:08048428                 jmp     short loc_8048454
.text:0804842A ; ---------------------------------------------------------------------------
.text:0804842A
.text:0804842A loc_804842A:                            ; CODE XREF: main+97j
.text:0804842A                 mov     eax, [esp+1Ch]
.text:0804842E                 shl     eax, 2
.text:08048431                 add     eax, [ebp+arg_4]
.text:08048434                 mov     edx, [eax]
.text:08048436                 mov     eax, offset aGoodbyeArgvDS ; "Goodbye! argv[%d] = %s\n"
.text:0804843B                 mov     [esp+8], edx
.text:0804843F                 mov     edx, [esp+1Ch]
.text:08048443                 mov     [esp+4], edx
.text:08048447                 mov     [esp], eax      ; format
.text:0804844A                 call    _printf
.text:0804844F                 add     dword ptr [esp+1Ch], 1
.text:08048454
.text:08048454 loc_8048454:                            ; CODE XREF: main+64j
.text:08048454                 mov     eax, [esp+1Ch]
.text:08048458                 cmp     eax, [ebp+arg_0]
.text:0804845B                 jl      short loc_804842A
.text:0804845D
.text:0804845D loc_804845D:                            ; CODE XREF: main+5Aj
.text:0804845D                 mov     eax, 0
.text:08048462                 leave
.text:08048463                 retn
```

```
.text:08048463 main            endp
.text:08048463

.. CODE SNIPPET ..

.rodata:08048528 ; ===========================================================================
.rodata:08048528
.rodata:08048528 ; Segment type: Pure data
.rodata:08048528 ; Segment permissions: Read
.rodata:08048528 _rodata         segment dword public 'CONST' use32
.rodata:08048528                 assume cs:_rodata
.rodata:08048528                 ;org 8048528h
.rodata:08048528                 public _fp_hw
.rodata:08048528 _fp_hw          dd 3
.rodata:0804852C                 public _IO_stdin_used
.rodata:0804852C _IO_stdin_used  dd 20001h
.rodata:08048530                 public __dso_handle
.rodata:08048530 __dso_handle    db    0
.rodata:08048531                 db    0
.rodata:08048532                 db    0
.rodata:08048533                 db    0
.rodata:08048534 ; char format[]
.rodata:08048534 format          db 'Hello! argv[%d] = %s',0Ah,0 ; DATA XREF: main+33o
.rodata:0804854A ; char aGoodbyeArgvDS[]
.rodata:0804854A aGoodbyeArgvDS  db 'Goodbye! argv[%d] = %s',0Ah,0 ; DATA XREF: main+72o
.rodata:0804854A _rodata         ends
.rodata:0804854A

.. CODE SNIPPET
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *IEEE Symposium on Security and Privacy (SP), 2010*. IEEE Computer Society, May 2010.

[2] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, pp. 385–394, Jul. 1976.

[3] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008, pp. 206–215.

[4] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Sage: Whitebox fuzzing for security testing,” *ACM Queue*, Jan. 2012. [Online]. Available: http://queue.acm.org/detail.cfm?id=2094081

[5] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in *Proceedings of the 15th Annual Network and Distributed System Security Conference*. ACM, 2008.

[6] P. Godefroid, “Random testing for security: Blackbox vs. whitebox fuzzing,” in *Proceedings of the 2nd International Workshop on Random Testing: Co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 1–1.

[7] S. D. Suh and I. Neamtiu, “Studying software evolution for taming software complexity,” in *Proceedings of the 2010 21st Australian Software Engineering Conference*. IEEE Computer Society, 2010, pp. 3–12.

[8] E. L. A. Lab and D. Harley, “Trends for 2011: Botnets and dynamic malware,” Jan. 2011, white Paper. [Online]. Available: http://go.eset.com/us/resources/white-papers/Trends-for-2011.pdf

[9] RSA, “The current state of cybercrime and what to expect in 2011,” 2011, rSA 2011 Cybercrime Trends Report.

[10] RSA, “The current state of cybercrime and what to expect in 2012,” 2012, rSA 2012 Cybercrime Trends Report.

[11] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag, 1999.

[12] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. No Starch Press, 2008.

[13] Hex-Rays, “Interactive disassembler (ida) pro.” [Online]. Available: http://www.hex-rays.com

[14] X. Chen, J. Anderson, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks*. IEEE Computer Society, Jun. 2008, pp. 177–186.

[15] M. Sikorski and A. Honig, *Practical Malware Analysis: A Hands on Guide to Dissecting Malicious Software*. No Starch Press, Jan. 2012.

[16] M. Prasad and T. Chiueh, “A binary rewriting defense against stack-based buffer overflow attacks,” in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Jun. 2003, pp. 211–224.

[17] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra, “An experimentation framework for evaluating disassembly and decompilation tools for c++ and java,” in *Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society, 2003, pp. 14–.

[18] A. Singh and B. Singh, *Identifying Malicious Code Through Reverse Engineering*. Springer-Verlag, 2009.

[19] VMProtect, “Vmprotect,” 2012. [Online]. Available: http://vmpsoft.com/

[20] M. Oberhumer, L. Molnar, and J. Reiser, “The ultimate packer for executables,” 2009. [Online]. Available: http://upx.sourceforge.net

[21] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, 1997.

[22] T. L. Team, “The llvm compiler infrastructure,” Jan. 2012. [Online]. Available: http://www.llvm.org

[23] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis,” in *Proceedings of the 4th International Conference on Information Security*. Springer-Verlag, Dec. 2008.

[24] C. Cadar, D. Dunbar, and D. R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.

[25] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler, “Automatically generating malicious disks using symbolic execution,” in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006, pp. 243–257.

[26] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[27] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. ACM, 1971, pp. 151–158.

[28] M. Weiser, “Program slicing,” in *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, 1981, pp. 439–449.

[29] M. Davis, G. Logemann, and D. Loveland, “A machine program for therem-proving,” *Communications of the ACM*, vol. 5, pp. 394–397, Jul. 1962.

[30] S. Bardin and P. Herrmann, “Structural testing of executables,” in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society, 2008, pp. 22–31.

[31] V. Developers, “Valgrind dynamic instrumentation and analysis,” 2012. [Online]. Available: http://www.valgrind.org/

[32] Q. Developers, “Qemu: Open source processor emulator,” 2012. [Online]. Available: http://www.qemu.org

[33] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 2006, pp. 322–335.

[34] V. Ganesh and D. L. Dill, "A decision producer for bit-vectors and arrays," in *Proceedings of the International Conference in Computer Aided Verification*. Springer-Verlag, Jul. 2007, pp. 519–531.

[35] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86 - a platform for analyzing x86 executables," in *Proceedings of the 14th International Conference on Compiler Construction*. Springer-Verlag, 2005, pp. 250–254.

[36] T. Reps, G. Balakrishnan, and J. Lim, "Intermediate-representation recovery from low-level code," in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 2006, pp. 100–111.

[37] Grammatech, "Codesurfer/x86," 2012. [Online]. Available: http://www.grammatech.com/research/products/CodeSurferx86.html/

[38] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analyses of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977, pp. 238–252.

[39] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automatated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 213–223.

[40] Intel and U. of Virginia, "Pin: A dynamic binary instrumentation tool," 2012. [Online]. Available: http://www.pintool.org

[41] Hewlett-Package and M. I. of Technology, "Dynamorio: Dynamic instrumentation tool platform," 2012. [Online]. Available: http://www.dynamorio.org

[42] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[43] M. Ganai and A. Gupta, *SAT-Based Scalable Formal Verification Solutions*, ser. Integrated Circuits and Systems. Springer-Verlag, 2007.

[44] S. Qadeer, “Algorithms and methodology for scalable model checking,” Ph.D. dissertation, University of California, Berkeley, 1999.

[45] W. J. Yeh, “Controlling state explosion in reachability analysis,” Ph.D. dissertation, Purdue University, 1993.

[46] Z. Xing-feng, W. Jian-dong, L. Bin, Z. Jun-wu, and W. Jun, “Methods to tackle state explosion problem in model checking,” in *Proceedings of the Third Symposium on Intelligence Information Technology Application*. IEEE Press, Nov. 2009, pp. 329–331.

[47] P. Boonstoppel, C. Cadar, and D. R. Engler, “Rwset: Attacking path explosion in constraint-based test generation,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2008, pp. 351–366.

[48] P. Godefroid and S. Khurshid, “Exploring very large state spaces using genetic algorithms,” in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2002, pp. 266–280.

[49] X. Li, D. Shannon, I. Ghosh, M. Ogawa, S. P. Rajan, and S. Khurshid, “Context-sensitive relevancy analysis for efficient symbolic execution,” in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. Springer-Verlag, 2008, pp. 36–52.

[50] P. Godefroid, G. J. Holzmann, and D. Pirottin, “State-space caching revisited,” in *Proceedings of the Fourth International Workshop on Computer Aided Verification*. Springer-Verlag, 1993, pp. 178–191.

[51] R. Pelánek, *Fighting State Space Explosion: Review and Evalution*. Springer-Verlag, 2009, pp. 37–52.

[52] R. J. Schonberger, *Japanese Manufacturing Techniques: Nine Hidden Lessons in Simplicity*. Free Press, 1982.

[53] T. Reinbacher and J. Brauer, “Precise control flow reconstruction using boolean logic,” in *Proceedings of the Ninth ACM International Conference on Embedded Systems*. ACM, 2011, pp. 117–126.

[54] J. Newsome and D. Song, “Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software,” in *Proceedings of the Network and Distributed Systems Security Symposium*. The Internet Society, Feb. 2005.

[55] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 135–148.

[56] L. de Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Formal Methods: Foundations and Applications*, ser. 12th Brazilian Symposium on Formal Methods, M. V. Oliveira and J. Woodcock, Eds. Springer-Verlag, 2009, pp. 23–36.

[57] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, 1st ed. Springer-Verlag, 2008.

[58] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, 2007.

[59] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, pp. 201–215, Jul. 1960.

[60] T. Fahringer and B. F. Scholz, "A unified symbolic evaluation framework for parallelizing compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 11, pp. 1105–1125, Nov. 2000.

[61] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 1, 2A, 2B, 3A, and 3B*. Intel, 2011.

[62] G. S. Tseitin, "On the complexity of derivation in the propositional calculus," *Studies in Constructive Mathematics and Mathematical Logic*, vol. 2, pp. 115–215, 1968.

[63] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman, "Efficiently computing static single assignment form and the control dependencies graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[64] J. Ferreiros, "The road to modern logic: An interpretation," *The Bulletin of Symbolic Logic*, vol. 7, no. 4, pp. 441–484, 2001.

[65] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[66] S.-L. Authors, "Smt-lib: The satisfiability modulo theories library," 2012. [Online]. Available: http://www.smtlib.org/

[67] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard," Dec. 2010, version 2.0. [Online]. Available: http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.12.21.pdf

[68] D. R. Cok, "The smt-lib v2 language and tools," Feb. 2011, tutorial. [Online]. Available: http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf

[69] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady, "Deciding bit-vector arithmetic with abstraction," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2007, pp. 358–372.

[70] K. R. M. Leino and F. Logozzo, "Loop invariants on demand," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*. Springer-Verlag, 2005, pp. 119–134.

[71] P. Cousot, "Abstract interpretation based formal methods and future challenges," in *Informatics - 10 Years Back. 10 Years Ahead.* Springer-Verlag, 2001, pp. 138–156.

[72] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing enginer for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with th 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2005, pp. 263–272.

[73] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillman, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1066–1071.

[74] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 24–44.

[75] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *Runtime Verification, 9th International Workshop*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2009, pp. 76–92.

[76] B. Team, "Vine installation and user manual," Aug. 2009. [Online]. Available: http://bitblaze.cs.berkeley.edu/release/vine-1.0/howto.pdf

[77] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics - 10 Years Back. 10 Years Ahead.* Springer-Verlag, 2001, pp. 176–194.

[78] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *Proceedings of the 1991 International Conference on Very Large Scale Integration*. North-Holland, Aug. 1991.

[79] G. Diamos, "State explosion: An obvious limitation to strong scaling," Sep. 2009, short Paper.

[80] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based cfg reconstruction from unstructured programs," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2011, pp. 54–69.

[81] T. Fahringer and B. F. Scholz, *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization*. Springer-Verlag, 2003.

[82] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, Sep. 1986.

[83] M. Corporation, "Z3 theorem prover," 2012. [Online]. Available: http://research.microsoft.com/en-us/um/redmond/projects/z3/

Referenced Authors

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California